

Análise e design orientados a objetos para sistemas de informação

Modelagem com UML, OCL
e IFML

Raul Sidnei Wazlawick



VISIT...

LANZAROTE
Caliente.COM

© 2015, Elsevier Editora Ltda.

Todos os direitos reservados e protegidos pela Lei n. 9.610, de 19/02/1998.

Nenhuma parte deste livro, sem autorização prévia por escrito da editora, poderá ser reproduzida ou transmitida sejam quais forem os meios empregados: eletrônicos, mecânicos, fotográficos, gravação ou quaisquer outros.

Revisão: Flor de Letras Editorial

Editoração Eletrônica: SBNigri Artes e Textos Ltda.

Elsevier Editora Ltda.

Conhecimento sem Fronteiras

Rua Sete de Setembro, 111 – 16º andar

20050-006 – Centro – Rio de Janeiro – RJ – Brasil

Rua Quintana, 753 – 8º andar

04569-011 – Brooklin – São Paulo – SP – Brasil

Serviço de Atendimento ao Cliente

0800-0265340

sac@elsevier.com.br

ISBN 978-85-352-7984-9

Nota: Muito zelo e técnica foram empregados na edição desta obra. No entanto, podem ocorrer erros de digitação, impressão ou dúvida conceitual. Em qualquer das hipóteses, solicitamos a comunicação ao nosso Serviço de Atendimento ao Cliente, para que possamos esclarecer ou encaminhar a questão.

Nem a editora nem o autor assumem qualquer responsabilidade por eventuais danos ou perdas a pessoas ou bens, originados do uso desta publicação.

CIP-Brasil. Catalogação na fonte.
Sindicato Nacional dos Editores de Livros, RJ

W372a

Wazlawick, Raul Sidnei, 1967-

Análise e projeto de sistemas de informação orientados a objetos:
modelagem com UML, OCL e IFML / Raul Wazlawick. – [3. ed.] – Rio
de Janeiro: Elsevier, 2015.
il. ; 24 cm.

ISBN 978-85-352-7984-9

1. Administração de projetos. 2. Engenharia de software. 3. Software
– Desenvolvimento. I. Título.

14-17370.

CDD: 658.404

CDU: 005.8

A meus pais e ancestrais, sem os quais eu não existiria.

Agradecimentos

Quero agradecer a algumas pessoas que, de uma forma ou de outra, ajudaram a tornar este livro possível: Professor Luiz Fernando Bier Melgarejo, por me iniciar no mundo dos sistemas orientados a objetos já em 1987; meu colega Professor Marcos Eduardo Casa, por todo o trabalho de programação em pares feito ainda nos anos de 1980, quando a orientação a objetos era “coisa de outro mundo”; meu colega Professor Antonio Carlos Mariani, por desenvolver o Mundo dos Atores, uma ferramenta que nos ajudou a ensinar programação orientada a objetos; as empresas que me permitiram usar as técnicas explicadas neste livro em ambientes reais de desenvolvimento; meus orientandos Everton Luiz Vieira, Kuesley Fernandes do Nascimento e Iuri Cardoso, por ajudarem a consolidar algumas das novas técnicas; o Departamento de Informática e Estatística (INE) da Universidade Federal de Santa Catarina (UFSC), pela oportunidade de desenvolver este trabalho; e especialmente ao engenheiro de software Gilmar Purim, pelas excelentes discussões que ajudaram a definir o formato da primeira edição deste livro em 2003.

Também agradeço aos mais de mil alunos, que não posso citar nominalmente, vítimas de minha disciplina de Análise e Projeto Orientados a Objetos por mais de duas décadas. Suas dúvidas, erros e dificuldades me motivaram a aprender ainda mais; a meu amigo, Professor Rogério Cid Bastos, pela orientação e motivação; e finalmente a meus amigos e irmãos, pelos momentos de alegria e fraternidade.

Prefácio

Porque as coisas são frequentemente difusas no início de um projeto de software, porque muitas vezes é difícil fazer os interessados concordarem, porque as informações e funções necessárias muitas vezes não são claras, porque os clientes não se sentem culpados por solicitar sistemas com complexidade e tamanho assustadores e porque é tão tentador assumir que já sabemos o que é solicitado, resta uma necessidade urgente de bons livros que ensinem aos engenheiros de software uma abordagem pragmática e testada para análise e design. Raul Wazlawick nos fornece este livro.

Nos últimos 50 anos, vimos muitos métodos propostos para análise e design de software computacional, mas, na minha visão, os mais intuitivos e efetivos adotam a mentalidade orientada a objetos. Lembre-se, análise e design ocorrem relativamente cedo no processo de software, em um tempo em que as coisas são fluidas, a iteração não é apenas comum, mas obrigatória, e os “objetos” são frequentemente os elementos mais visíveis do problema.

A beleza do paradigma orientado a objetos está na sua maravilhosa elasticidade. Nós podemos descrever objetos que são facilmente entendidos pelo pessoal de negócios no nível conceitual e, então, por meio de um processo de iteração e elaboração, refinar tais objetos para níveis de abstração bem mais baixos, que o pessoal técnico pode usar efetivamente. Neste livro, o Professor Wazlawick nos fornece um guia para conseguir isso.

Usando UML como sua forma notacional, ele inicia onde todos os bons analistas deveriam iniciar – no começo! Ele nos fornece as ferramentas necessárias para entender o problema por meio de um processo de eliciação, elaboração e representação. Ele enfatiza casos de uso e ajuda a entender que eles servem tanto como um excelente mecanismo para entendimento quanto como um mecanismo que pode ajudar bastante a auxiliar o planejamento de projeto.

Ele insiste que bons analistas e bons designers devem iterar à medida que desenvolvem um entendimento crescentemente melhor e mais detalhado do problema. Para conseguir isso, Wazlawick fornece ao leitor técnicas orientadas a objetos e ferramentas da UML necessárias para fazer o trabalho.

Por meio do livro, um estudo de caso é usado para fornecer exemplos de todas as técnicas de análise e design. Princípios básicos são introduzidos, expandidos em um conjunto de técnicas trabalháveis e, então, usados para projetar representações e modelos usando UML.

Durante minha longa carreira profissional, eu me encontrei com centenas de empresas e milhares de profissionais de software ao redor do mundo. Cada empresa e cada pessoa têm uma opinião única sobre o que faz o software tão desafiador, e há muitas opiniões sobre como resolver esses desafios em uma situação particular. Mas poucos profissionais

de software discordariam de uma simples afirmação: “Se você não sabe para onde está indo, é muito, muito difícil chegar lá”.

Métodos efetivos de análise e design dão aos profissionais de software um destino e um mapa para chegar lá. Em *Análise e design de sistemas de informação orientados a objetos*, Raul Wazlawick fornece as ferramentas de que os profissionais de software precisam para desenhar o mapa da estrada e, como consequência, este livro ajuda qualquer leitor a chegar lá onde ele precisa ir.

Roger S. Pressman, Ph. D.

Prefácio do Autor

Este livro apresenta de forma didática e aprofundada elementos de análise e design orientados a objetos para o desenvolvimento de sistemas de informação.

A área de desenvolvimento de software para sistemas e informação tem evoluído nas últimas décadas ao redor da Linguagem de Modelagem Unificada (UML) e do Processo Unificado (UP), adotados como padrões internacionais pelo *Object Management Group* (OMG). Mesmo os métodos de desenvolvimento ágil atualmente utilizam orientação a objetos como o paradigma dominante.

Em contraste com outros livros nesta área, que são organizados ao redor da apresentação dos diagramas da UML e seus muitos usos possíveis, este livro se concentra nas atividades diárias de analistas e designers, e explica como os diferentes diagramas podem ajudá-los a desenvolver sistemas melhores e a desenvolver melhor os sistemas.

Embora a cultura de orientação a objetos tenha mais de 40 anos de existência, ainda há empresas que acham difícil desenvolver software de boa qualidade, especialmente em razão da falta de manutenibilidade e flexibilidade nos sistemas que elas produzem. Este livro apresenta informações e técnicas para minimizar tal deficiência.

As técnicas descritas aqui foram aplicadas com sucesso pelo autor em muitos projetos de desenvolvimento de software em empresas, bem como em projetos de pesquisa em universidades.

O livro é voltado para profissionais de computação (analistas, designers e programadores) e estudantes de computação, tanto em nível de graduação quanto de pós-graduação, que tenham pelo menos um conhecimento mínimo em desenvolvimento ou modelagem orientada a objetos. Iniciantes e profissionais com experiência encontrarão nesta obra um guia útil para melhorar a qualidade de seus projetos. A intenção é ajudá-los a obter o melhor design possível.

Em contraste com o que ocorre com o modelo Waterfall (Royce, 1970), no qual as atividades de análise e design ocorrem sequencialmente, modelos iterativos como o UP propõem que essas atividades (também chamadas de *disciplinas*) possam ocorrer com maior ou menor ênfase em qualquer fase.

Este livro apresenta mais ênfase nas atividades que são tipicamente realizadas por analistas e designers, incluindo modelagem de negócio, engenharia de requisitos, análise e design. Outras disciplinas como gerenciamento de projetos, implementação e teste também são referenciadas. Para mais informações sobre outras disciplinas, livros de engenharia de software, como o de Pressman (2010), podem ser consultados, ou ainda livros mais específicos sobre o *Processo unificado*, como o de Kruchten (2003).

O objetivo da fase de *Concepção* do UP é construir uma visão geral do sistema e do seu contexto e planejar o desenvolvimento futuro. As duas principais ferramentas de modelagem para esta fase são o *modelo conceitual* (Capítulos 6 e 7) e os *casos de uso de alto nível* (Capítulo 3). A *modelagem de negócio* (Capítulo 2) pode ser necessária com maior ou menor ênfase, dependendo do projeto que será desenvolvido.

No final da fase de *Concepção*, casos de uso de alto nível serão utilizados como uma base para o *planejamento* do resto do projeto (Capítulo 4). Usualmente, este planejamento envolve avaliar os casos de uso como pequenos, médios ou grandes (escala *camiseta*) em relação à sua complexidade, e decidir sobre sua prioridade. Quando a complexidade de cada caso de uso é conhecida, é possível estimar o tempo médio necessário para desenvolvê-los (Karner, 1993) e, assim, as iterações podem ser planejadas como um conjunto de atividades que podem ser realisticamente realizadas no tempo disponível.

A fase de *Elaboração* do UP é realizada como uma sequência de iterações. Cada iteração possui um conjunto de casos de uso atribuídos para desenvolvimento¹. O desenvolvimento de um caso de uso individual normalmente inicia com sua expansão em fluxos (Capítulo 5). Opcionalmente, os fluxos de um caso de uso também podem ser representados como *diagramas de sequência de sistema*, com o objetivo de ajudar a descobrir as consultas e os comandos que devem ser implementados.

Durante a fase de *Elaboração*, quando a arquitetura está evoluindo, o *modelo conceitual* (Capítulos 6 e 7) é refinado e completado.

Consultas e comandos de sistema podem ser implementados com uma linguagem de programação. Mas o designer pode escolher, em vez disso, especificar *contratos* para consultas e comandos de sistema (Capítulo 8), os quais, além de serem definições de alto nível para o comportamento esperado de cada procedimento, podem ser muito úteis para definir *casos de teste* (Capítulo 11). Estes contratos correspondem ao modelo funcional do sistema, porque definem a funcionalidade de cada consulta e comando em termos das entradas que recebem e dos resultados que produzem.

Neste ponto, novamente o sistema pode ser implementado, ou seu código pode ser automaticamente gerado, se ferramentas estiverem disponíveis. Entretanto, caso deseje código elegante, pode ser útil produzir um novo conjunto de diagramas (diagramas de sequência ou comunicação) para cada uma das operações de sistema (Capítulo 9). Tais diagramas indicam como os diferentes objetos trocarão mensagens para distribuir responsabilidades e obter as pós-condições desejadas especificadas pelos contratos. Com este procedimento, padrões de projeto são mais fáceis de aplicar ao código. Ademais, esses diagramas também ajudam a especificar quais métodos (além de *getters* e *setters*) devem ser implementados em cada classe e como isso deve ser feito.

¹ Uma iteração também pode ter um *plano de mitigação* atribuído para um dado risco ou uma *solicitação de mudança* originada pelo cliente, usuário ou mesmo pela equipe de desenvolvimento.

Durante a Elaboração, o design da interface pode ser completado (Capítulo 12). Este livro apresenta a notação *Information Flow Modeling Language* (IFML)² como uma opção para modelar este aspecto do sistema. A IFML é uma nova linguagem de modelagem recentemente (março de 2013) adotada como padrão pelo OMG.

Embora isso seja mais enfático durante a fase de Construção, a Elaboração também pode precisar de geração de uma base de dados (Capítulo 13) e código executável (Capítulo 10). A persistência de dados pode ser gerada automaticamente se ferramentas adequadas estiverem disponíveis. Entretanto, vale a pena saber pelo menos como o mecanismo de persistência funciona, mesmo que seja automatizado. A geração de código é também apresentada como um conjunto de regras que podem ser automatizadas.

Para economizar espaço para uma apresentação mais profunda dos tópicos, a maioria dos exercícios poderá ser acessada em www.elsevier.com.br.

² Disponível em: <www.ifml.org>.

SIGLAS

API	Application Program Interface – Interface de Programação de Aplicações
AUP	Agile Unified Process – Processo Unificado Ágil
BPMN	Busines Process Model and Notation – Notação e Modelo de Processo de Negócio
CASE	Computer Aided Software Engineering – Engenharia de Software Auxiliada por Computador
CEP	Código de Endereçamento Postal
CMMI	Capability Maturity Model Integration – Integração de Modelos de Capacidade e Maturidade
COCOMO II	Constructive Cost Model II – Modelo de Custo Construtivo II
COTS	Commercial off the Shelf – Produto de Prateleira
CPF	Cadastro de Pessoa Física
CQS	Command-Query Separation – Separação Comando-Consulta
CRU	Create, Retrieve, and Update – Inserir, Consultar e Atualizar
CRUD	Create, Retrieve, Update, and Delete – Inserir, Consultar, Atualizar e Remover
CRUDL	Create, Retrieve, Update, Delete, and List – Inserir, Consultar, Atualizar, Remover e Listar
DBMS	Database Management System – Sistema Gerenciador de Banco de Dados
DCD	Diagrama de Classes de Design
DDD	Discagem Direta a Distância
EBP	Elementary Business Process – Processo Elementar de Negócio
EF	Environmental Factor – Fator Ambiental
FK	Foreign Key – Chave Estrangeira
FPA	Function Point Analysis – Análise de Pontos de Função
FURPS+	Functionality, Usability, Reliability, Performance , and Supportability – Funcionalidade, Usabilidade, Confiabilidade, Desempenho e Suportabilidade.
FUSP	Fuzzy Use Case Points – Pontos de Caso de Uso Difusos
IFML	Interaction Flow Modeling Language – Linguagem de Modelagem de Fluxo de Interação
INE	Informática e Estatística (Departamento)
ISBN	International Standard Book Number – Número de Livro Padrão Internacional
ISO/IEC	International Organization for Standardization / International Electro-technical Commission – Organização Internacional de Padronização / Comissão Eletrotécnica Internacional
KSLOC	Kilo Source Lines of Code – Milhares de Linhas de Código Fonte
MK II	Mark II
MoSKoW	Must, Should, Could, or Would – Deve, Deveria, Pode ou Poderia

MVC	Model-View-Controller – Modelo-Visão-Controlador
MVP	Model-View-Presenter – Modelo-Visão-Apresentador
NN	Not Null – Não Nulo
OCL	Object Constraint Language – Linguagem de Restrições sobre Objetos
OID	Object Identifier – Identificador de Objeto
OMG	Object Management Group – Grupo de Gerência de Objetos
OpenUP	Open Unified Process – Processo Unificado Aberto
ORM	Object-Relational Mapping – Mapeamento Objeto-Relacional
PK	Primary Key – Chave Primária
ROI	Return of Investment – Retorno de Investimento
RUD	Retrieve, Update, and Delete – Consultar, Atualizar e Remover
SCED	Required Development Schedule – Cronograma de Desenvolvimento Requerido
SCRUD	Search, Create, Retrieve, Update, and Delete – Pesquisar, Inserir, Consultar, Atualizar e Remover
SEI-CMU	Software Engineering Institute-Carnegie-Mellon University – Instituto de Engenharia de Software-Universidade Carnegie Mellon
SI	Sistemas de Informação
SLOC	Source Lines of Code – Linhas de Código Fonte
SMS	Short Message Service – Serviço de Mensagens Curtas
SPICE	Software Process Improvement and Capability Determination – Melhoria de Processo de Software e Determinação de Capacidade
SQL	Structured Query Language – Linguagem Estruturada de Consulta
TCF	Technical Complexity Factor – Fator de Complexidade Técnica
TCP/IP	Transmission Control Protocol/Internet Protocol – Protocolo de Controle de Transmissão/Protocolo de Internet
TDD	Test Driven Development – Desenvolvimento Dirigido pelo Teste
TLC	Team Load Capacity – Capacidade de Carga da Equipe
TPI	Team Productivity Index – Índice de Produtividade da Equipe
UAW	Unadjusted Actor Weight – Peso de Atores não Ajustado
UCP	Use Case Points – Pontos de Caso de Uso
UFSC	Universidade Federal de Santa Catarina
UKSMA	United Kingdom Software Metrics Association – Associação Britânica de Métricas de Software
UML	Unified Modeling Language – Linguagem de Modelagem Unificada
UP	Unified Process – Processo Unificado
USC	University of South California – Universidade do Sul da Califórnia
UUCP	Unadjusted Use Case Points – Pontos de Caso de Uso não Ajustados
UUCW	Unadjusted Use Case Weight – Peso de Casos de Uso não Ajustado
XML	Extensible Markup Language – Linguagem de Marcação Extensível
XP	Extreme Programming – Programação Extrema
XSL	Extensible Stylesheet Language – Linguagem Extensível de Folha de Estilos

Introdução

TÓPICOS-CHAVE NESTE CAPÍTULO

- Desenvolvimento de sistemas orientados a objetos
 - Linguagem de Modelagem Unificada (UML)
 - Processo Unificado (UP)
-

1.1 Este livro

Existe uma vasta literatura que procura apresentar sintaticamente os diagramas da Linguagem de Modelagem Unificada (UML)¹, como, por exemplo, Miles e Hamilton (2006). Há também um número significativo de livros sobre processo de desenvolvimento e atividades de gerência, tais como o de Satzinger, Jackson e Burd (2011). Entretanto, existem relativamente poucos livros que vão fundo na apresentação das melhores práticas que permitem a aplicação efetiva das técnicas orientadas a objetos no desenvolvimento de software no mundo real. Isso significa que, apesar da vasta literatura nesta área, muitas questões com as quais os desenvolvedores se deparam ainda não são respondidas.

Este livro apresenta mais detalhes sobre vários conceitos e atividades da orientação a objetos, especialmente aqueles introduzidos por Larman (2004), que são baseados no Processo Unificado, mais comumente conhecido como UP (Jacobson; Booch; Rumbaugh, 1999; Scott, 2001).

A razão para este livro ser baseado nos conceitos de Larman (2004) é que ele apresenta uma abordagem concisa e eficiente para análise e design de sistemas usando o paradigma orientado a objetos. Em sua abordagem, cada artefato (documento ou diagrama) tem uma razão clara e prática para existir, e as conexões entre os artefatos são muito precisas.

Em nossa abordagem, princípios ágeis adotados pela *Programação Extrema* (XP) (Beck; Andres, 2004) são largamente aceitos. Entretanto, nós propomos que a programação deve ser feita primeiro em uma linguagem de modelagem antes de programar o código diretamente. O uso dos diagramas da UML como ferramenta de programação de alto nível, em vez de simples ferramenta de documentação, é também recomendado. Com esta abordagem, os artefatos usados devem contribuir diretamente para o entendimento do problema (análise) ou para o design da solução. Para melhorar o processo de projetar uma solução, nós consideramos que a transformação de modelos e a geração automática de código são fundamentais.

¹ Disponível em: <www.uml.org>.

Além disso, em contraste com o trabalho de Larman, este livro apresenta alguns conceitos originais e novos detalhes em tópicos, tais como:

- Critérios objetivos para identificar casos de uso e decidir quando subdividi-los ou não.
- Uma técnica para expandir casos de uso que reduz a disparidade entre as descrições criadas por diferentes analistas (os conceitos de passos obrigatórios e complementares permitem decidir exatamente quantos passos um caso de uso deve ter).
- Diagramas de sequência de sistema construídos com atores, interface e controle (em vez de apenas atores e sistema) para ajudar a perceber a clara diferença entre eventos de sistema e operações de sistema.
- Uma abordagem original para escrever contratos para comandos e consultas de sistema com o uso da *Object Constraint Language* (OCL) (Object Management Group, 2010), que permite a geração automática de código executável, não apenas checagem de pós-condições, como as ferramentas atuais fazem.
- Adaptação dos padrões de análise de Fowler (2003) para UML e reestruturação de alguns desses padrões para simplificar sua identificação e aplicação na prática.
- Uma técnica sistemática para gerar modelos dinâmicos de objetos a partir de contratos OCL, os quais seguem bons padrões de projeto e reduzem significativamente a quantidade de código necessária para rodar uma aplicação, evitando, por exemplo, verificações redundantes.
- Um design de camada de interface apresentado com o uso de *Interaction Flow Modeling Language* (IFML), um novo padrão da *Object Management Group* (OMG) para modelagem de interfaces.

As características antes mencionadas são úteis para produzir software de alta qualidade, que seja bem organizado, baseado em uma arquitetura multicamadas e capaz de acomodar mudanças ou novos requisitos.

1.2 Desenvolvimento de sistemas orientados a objetos

O que é desenvolvimento de sistemas orientados a objetos? Observando a maneira como análise e design são ensinados e praticados em alguns lugares, pode-se concluir que muitos profissionais simplesmente adotam linguagem e programação orientadas a objetos ou usam fragmentos de algum processo de desenvolvimento orientado a objetos. Entretanto, com muita frequência, eles não são tão efetivos quanto se poderia esperar.

Não é suficiente organizar a arquitetura de um sistema em camadas e módulos se o código implementado dentro deles for desorganizado. Alguns desenvolvedores organizam o sistema adequadamente em classes e pacotes, mas eles ainda escrevem *código espagete* dentro dos métodos dessas classes e pacotes. Adicionalmente, outros desenvolvedores ainda usam decomposição funcional *top-down* dentro dos métodos, o que não é adequado

quando se usa programação orientada a objetos (decomposição *top-down* seria adequada caso fosse usada programação estruturada).

Para construir código que seja realmente orientado a objetos, os desenvolvedores devem aprender as técnicas de *delegação* e *atribuição de responsabilidades*, as quais podem levar à construção de código reusável e com baixo acoplamento. Essas técnicas são explicadas neste livro.

É inútil investir de forma pesada em ferramentas *Computer-Aided Software Engineering* (CASE) orientadas a objetos sem aprender a *pensar* em termos de programação orientada a objetos. O uso de diagramas não necessariamente vai melhorar a qualidade do software, embora possa ajudar.

1.3 Linguagem de Modelagem Unificada (UML)

Alguns desenvolvedores acreditam que UML seja uma metodologia, talvez por causa do “M” na sigla. Entretanto, isso não é verdade: UML significa, em inglês, *Unified Modeling Language* (Linguagem de Modelagem Unificada), portanto, é uma *linguagem* que pode ser usada para descrever coisas.

Conhecer uma linguagem não implica necessariamente a habilidade de produzir artefatos úteis com ela. Por exemplo, a língua portuguesa é uma linguagem, mas alguém que saiba falar português não necessariamente saberá escrever boa poesia ou fazer belos discursos. Além da sintaxe da linguagem, existem o conhecimento e as técnicas relacionados às melhores práticas que ajudam bastante aos poetas e oradores a colocar os elementos da linguagem em uma ordem e estrutura que sejam adequadas para produzir os resultados esperados.

A linguagem UML tem sido desenvolvida desde que James Rumbaugh e Grady Booch juntaram forças na empresa Rational Software e começaram a unificar suas notações diagramáticas e processos já bastante conhecidos. Ivar Jacobson juntou-se ao grupo e adicionou seus casos de uso e outras notações à linguagem unificada que já estava em desenvolvimento.

A UML é revisada constantemente e atualmente tem as seguintes três famílias de diagramas:

- *Diagramas estruturais*: incluindo diagramas de pacotes, classes, objetos, estrutura composta, componentes, perfil e implantação. Eles são usados para definir o que deve ser implementado no sistema em termos de componentes. Eles são úteis para especificar a parte da arquitetura do sistema que é independente do tempo.
- *Diagramas comportamentais*: incluindo diagramas de casos de uso, atividades e máquinas de estado. Eles enfatizam o que deve acontecer no sistema ou processo de negócio. Eles são usados para descrever a funcionalidade do sistema.

- *Diagramas de interação*: incluindo diagramas de comunicação, sequência, tempo e visão geral de interação. Eles são um subconjunto dos diagramas comportamentais e descrevem os fluxos de controle entre diferentes componentes do sistema.

Nem todos os diagramas devem ser usados durante o desenvolvimento de um sistema. Apenas aqueles que representam informações úteis para o projeto são recomendados. Esta obra enfatiza o uso dos diagramas de atividade, máquina de estados, casos de uso, sequência, comunicação e classes para modelar sistemas de informação. Entretanto, outros diagramas podem ser úteis dependendo das características do sistema sendo modelado. Para mais informação sobre os diferentes diagramas UML, o livro de Miles e Hamilton (2006) pode ser consultado.

1.4 Processo Unificado (UP)

As técnicas apresentadas neste livro são compatíveis com o Processo Unificado (UP), o qual é fortemente baseado (embora não necessariamente) em UML.

O UP também foi proposto pelos *três amigos*, Grady Booch, James Rumbaugh e Ivar Jacobson (Jacobson; Booch; Rumbaugh, 1999), como resultado de extensa experiência acumulada.

Este processo é baseado nos seguintes princípios:

- *Dirigido por casos de uso*: o desenvolvimento é planejado e organizado sobre uma lista de casos de uso.
- *Centrado na arquitetura*: o processo de desenvolvimento leva à construção de uma arquitetura de sistema que permite a implementação dos requisitos. Esta arquitetura é baseada na identificação de uma estrutura que é iterativamente construída a partir de um modelo conceitual.
- *Iterativo e incremental*: o desenvolvimento é dividido em iterações ou ciclos de desenvolvimento. A cada iteração, novas características são adicionadas à arquitetura do sistema, ou corrigidas/refinadas, deixando-a mais completa e mais parecida com a forma final do sistema desejado.
- *Orientado a riscos*: os elementos de maior risco para um projeto são tratados o mais cedo possível. Por exemplo, casos de uso críticos são identificados, detalhados e implementados antes dos demais.

O UP inclui em suas disciplinas as principais atividades relacionadas ao desenvolvimento de software. Estas atividades têm diferentes níveis ou ênfase durante as quatro grandes fases do UP: Concepção, Elaboração, Construção e Transição (Figura 1.1). Embora sequenciais no tempo, essas fases não devem ser confundidas com as fases do modelo *Waterfall* (Royce, 1970), no qual a especificação dos requisitos deve ser concluída antes do design; o design deve ser completado antes da construção, e assim por diante. Em UP, a especificação de requisitos, projeto, construção e outras atividades são realizadas em todas as fases com diferentes ênfases.

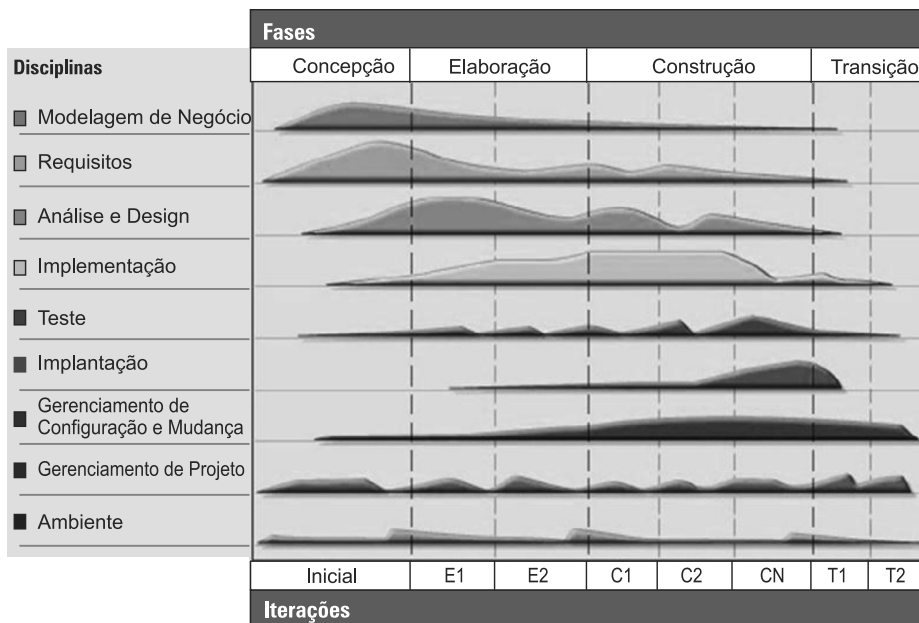


Figura 1.1

A ênfase das diferentes disciplinas durante as quatro fases do Processo Unificado Rational (RUP®)²⁻³.

Concepção é a primeira fase do UP, na qual os principais requisitos são descobertos e a extensão do sistema é compreendida. A saída desta fase, usualmente, consiste de um modelo conceitual preliminar (Seção 3.6); um documento de requisitos geralmente na forma de uma lista de casos de uso de alto nível (Seção 3.3) e especificações suplementares (Seção 3.5.8); e um cronograma de desenvolvimento baseado em uma lista de casos de uso (Seção 4.3). Adicionalmente, uma lista de riscos de alta exposição e seus planos de mitigação podem ser criados, bem como outros planos que atendem a necessidades especiais do projeto. Mas estes aspectos estão fora do escopo deste livro, o qual vai se concentrar nas técnicas de modelagem, e não em gerenciamento. Apenas o planejamento de iterações baseado em casos de uso de alto nível é apresentado em detalhes.

A fase de *Elaboração* inclui uma análise de requisitos mais detalhada, a qual é realizada por meio da expansão dos casos de uso, isto é, escrevendo uma sequência de passos que caracterizam cada um dos fluxos possíveis. O modelo conceitual é refinado após cada caso de uso ser expandido. Dependendo da prioridade dos casos de uso, espera-se que o número de mudanças aplicadas à arquitetura do software diminua à medida que o projeto avança pela fase de Elaboração.

² Rational Unified Process (RUP) é a mais antiga e mais conhecida implementação do Processo Unificado.

³ Disponível em: <www.ibm.com> Versão em português disponível em <http://mds.cultura.gov.br/minic/guidances/supportingmaterials/resources/humpchart.jpg>.

Durante a fase de *Construção* é realizada a maior parte das atividades de produção de código e teste. Espera-se que a fase de *Elaboração* produza uma arquitetura suficientemente estável, de forma que sua refatoração seja minimizada durante a *Construção*.

A fase de *Transição* consiste tipicamente nos testes finais e na entrega do sistema aos seus usuários, incluindo a possibilidade de sua instalação e migração de dados. Durante esta fase, o sistema será implantado, possivelmente substituindo um sistema existente (manual ou automatizado).

As fases de *Elaboração* e *Construção* são realizadas em iterações. Uma iteração deve ter como objetivo desenvolver um ou mais casos de uso, implementar mudanças solicitadas ou mitigar riscos selecionados. Durante uma iteração, casos de uso são expandidos e a informação que se aprende com eles é incrementalmente incorporada ao produto. Espera-se que a fase de *Elaboração* lide com os maiores riscos do projeto, bem como os casos de uso mais complexos ou arriscados que possam afetar a arquitetura do sistema de forma significativa. No entanto, a fase de *Construção* se concentra em produzir código para a aplicação completa e em implementar as mudanças solicitadas.

O UP é usualmente compreendido como um processo prescritivo, mas ele também pode ser realizado como um método ágil, com poucos artefatos. Duas implementações ágeis populares de UP são AUP⁴ (Ambler; Jeffries, 2002) e OpenUP⁵ (Kroll; MacIsaac, 2006). Um *método ágil* deve priorizar⁶:

- Pessoas e iterações acima de ferramentas e processos.
- Software funcionando acima de documentação compreensível.
- Colaboração do cliente acima de negociação de contrato.
- Responder à mudança acima de seguir um plano.

Para obter esta agilidade, toda documentação deve ser direcionada à produção de software. Cada atividade realizada pelo desenvolvedor deve ter um objetivo muito claro e um uso preciso, tendo como meta sempre a produção de código que satisfaz os requisitos da melhor forma possível e no menor tempo razoavelmente possível. O software é projetado com dois objetivos em mente: entender as necessidades do cliente e produzir uma solução viável que satisfaça essas necessidades. Para ajudar as pessoas a adequadamente comunicar suas necessidades e soluções, diferentes artefatos, como diagramas, podem ser criados; diagramas são mais úteis quando permitem que o código seja automaticamente gerado a partir deles.

1.5 O processo visto aqui

Ao final de cada capítulo, uma tabela mostrará quais atividades foram cobertas dentro de cada fase e disciplina do UP. O capítulo inicial é uma visão geral do UP. Apenas as fases de *Concepção* e *Elaboração* são cobertas neste livro, e somente as disciplinas de mode-

⁴ Disponível em: <www.ambysoft.com/unifiedprocess/agileUP.html>.

⁵ Disponível em: <epf.eclipse.org/wikis/openup/index.htm>.

⁶ Disponível em: <agilemanifesto.org>.

lagem de negócio, requisitos, análise e design, implementação, teste e alguns aspectos de gerenciamento de projetos são abordadas.

	Concepção	Elaboração	Construção	Transição
Modelagem de Negócio	Áreas cobertas por este livro			
Requisitos				
Análise e Design				
Implementação				
Teste				
Gerenciamento de Projeto				
Implementação				
Gerenciamento de Configuração e Mudança				
Ambiente				

1.6 Questões

1. Quais são os quatro princípios fundamentais do Processo Unificado?
2. Qual é o objetivo de cada uma das fases do Processo Unificado?
3. Em que as fases do Processo Unificado diferem das fases do modelo *Waterfall*?

Modelagem de negócio

2

TÓPICOS-CHAVE NESTE CAPÍTULO

- Visão geral do sistema
- Diagrama de casos de uso de negócio
- Diagrama de atividades
- Diagrama de máquina de estado

2.1 Introdução à modelagem de negócio

A fase de Concepção do UP consiste em um período no qual os analistas estão procurando obter informações sobre o negócio a ser automatizado ou reestruturado. Assume-se que o conhecimento que os analistas têm sobre o negócio é mínimo e que a interação com os interessados será intensa. O objetivo desta fase é descobrir se vale a pena fazer a análise, mas sem entrar muito profundamente no projeto.

Usualmente, antes de começar a descobrir e analisar os requisitos do sistema, é necessário entender os objetivos da organização-alvo. Esta visão pode ser obtida pela disciplina do UP chamada *modelagem de negócio*.

A modelagem de negócio é, portanto, uma atividade que suporta a descoberta dos requisitos de sistema porque ajuda a equipe a perceber o contexto mais amplo do negócio no qual o futuro sistema vai operar. Diferentes artefatos de texto e diagramas podem suportar essa disciplina, como este capítulo mostra.

Os artefatos nesta fase usualmente não precisam ser muito detalhados. Encontros com interessados podem produzir um registro (ata) que provavelmente pode referenciar várias ideias sobre as metas da empresa e oportunidades de informatização. Usando essa informação, a equipe pode construir uma visão geral do negócio.

Iniciando com os registros obtidos, a equipe pode organizar um documento inicial para o projeto, que pode ser chamado de *visão geral do sistema* ou *sumário executivo*. Este documento habitualmente inclui a *declaração de escopo*.

Outras informações relevantes são também frequentemente incluídas na visão geral, tais como os *interessados*, *atribuições de poder*, *análise de risco*, *orçamento e cronograma preliminares* e as *hipóteses do projeto*.

A declaração de escopo do projeto pode ser relacionada às atividades de análise de negócio. Após descobrir informações úteis sobre o negócio, a equipe será mais capaz de determinar o que deve e o que não deve ser incluído no projeto. A visão geral do sistema pode ser elaborada com a ajuda de alguns diagramas UML. Diagramas de caso de uso de

negócio (Seção 2.3) e diagramas de atividades de negócio (Seção 2.4) podem ser usados, e ocasionalmente, também diagramas de máquina de estado (Seção 2.5). Alternativamente, a *Business Process Model and Notation*¹ (BPMN)² pode ser usada no lugar dos diagramas de atividade da UML. Esses diagramas ajudam a determinar se o escopo do projeto é completo e consistente com as metas da organização.

A modelagem de negócio consiste em estudar e entender a organização e seus processos porque, habitualmente, o sistema que vai ser desenvolvido não é um produto isolado, mas parte orgânica de um contexto organizacional. Os objetivos desta disciplina são:

- Entender a *estrutura e a dinâmica* da organização-alvo na qual o software será usado.
- Entender os problemas atuais da organização-alvo e identificar *melhorias potenciais* que possam ser obtidas com o software.
- Garantir que clientes, usuários e a equipe de desenvolvimento compartilhem um *entendimento consistente* sobre a organização-alvo.
- Derivar os *requisitos* para obter as melhorias desejadas.

Um dos pontos críticos para o sucesso de um projeto de desenvolvimento de software é o seu financiamento. Para manter o interesse no desenvolvimento de um produto de software, um cliente deve permanecer convencido de que o investimento representa um ganho, e não um desperdício de tempo e dinheiro. A compreensão do modelo de negócio é fundamental para que a equipe possa manter o cliente interessado no projeto.

Outro aspecto importante da modelagem de negócio é aproximar a equipe de negócios da equipe de engenharia de software, de forma que os problemas da organização e suas necessidades sejam entendidos e resolvidos usando-se tecnologia da informação.

A modelagem de negócio é muito importante quando mudanças comportamentais significativas são introduzidas para um grupo de pessoas. Neste caso, o contexto de negócio e as consequências da instalação de um novo sistema devem ser conhecidos e estudados. A modelagem de negócio pode ser menos crítica quando tais mudanças comportamentais não são esperadas.

Assim, a modelagem de negócio pode ter diferentes níveis de importância, dependendo das necessidades de projeto. Kruchten (2003) identifica seis cenários de complexidade crescente em termos da necessidade de modelagem de negócio:

- *Organograma*: pode ser necessário apenas construir um organograma da organização para conhecer os setores e responsabilidades relacionados à área na qual o projeto está inserido. Neste caso, a modelagem de negócio, em geral, é muito simples e acontece apenas durante a Concepção. O organograma vai ajudar os analistas a localizar as pessoas que são responsáveis pelos diferentes aspectos relacionados ao futuro sistema e qual o grau de autoridade atribuído a cada um em relação a decidir sobre requisitos.
- *Modelagem de domínio*: se o objetivo do projeto é construir aplicações para gerenciar e apresentar informação, então a modelagem de negócio pode ser realizada durante

¹ Anteriormente conhecida como *Business Process Modeling Notation*.

² Disponível em: <www.bpmn.org>.

a análise de domínio, ou seja, quando um modelo estático da informação (modelo conceitual) é produzido. Neste caso, a modelagem de negócio é realizada durante a Concepção e a Elaboração.

- *Uma empresa, vários sistemas*: pode ser o caso de que a equipe esteja desenvolvendo toda uma família de sistemas para uma empresa. Neste caso, a modelagem de negócio não trata apenas de um sistema, mas se estende por vários projetos, e poderia inclusive ser um projeto independente. Ela ajudará a descobrir os requisitos dos sistemas individuais, bem como determinar uma arquitetura comum para toda a família.
- *Modelo de negócio genérico*: se o objetivo é construir um ou mais sistemas que servirão a um grupo de organizações, então a modelagem de negócio é útil para alinhar as diferentes organizações a uma visão de negócio comum ou, se isso não for possível, para obter uma visão de negócio na qual as especificidades das organizações sejam visíveis.
- *Novo negócio*: se uma organização decide iniciar um novo negócio, e todo um conjunto de sistemas deve ser desenvolvido para dar suporte a ele, então um esforço de modelagem de negócio significativo deve ser feito. Neste caso, a meta da modelagem de negócio não é apenas encontrar os requisitos, mas verificar a efetiva viabilidade do novo negócio. Assim, a modelagem de negócio, nessa situação, poderia ser um projeto independente.
- *Renovação*: se a organização decide renovar completamente sua forma de fazer negócios, então a modelagem de negócio será um projeto separado, realizado em várias etapas: visão do novo negócio; engenharia reversa do negócio existente; engenharia direta do novo negócio; e instalação do novo negócio.

Outros fatores que aumentam ou diminuem a necessidade da modelagem de negócio são:

- Requisitos bem definidos e estáveis no início do projeto (por exemplo, criar um substituto para um sistema existente) reduzem a necessidade de realizar uma modelagem de negócio intensa.
- Se o novo sistema vai mudar a forma como as pessoas fazem seu trabalho, então se deve prestar atenção especial à modelagem de negócio, especialmente quando o modelo de negócio vai mudar ou ser reconstruído após a automação de alguns casos de uso.
- Se o projeto tem riscos importantes, especialmente de negócio, então a modelagem de negócio deve ser mais detalhada e intensa.
- Se a comunidade de usuários é grande e heterogênea, então a modelagem de negócio provavelmente será mais difícil.
- Se for necessário integrar o novo sistema a hardware e sistemas legados, então a compreensão desses elementos deve ser incluída na modelagem de negócio, gerando mais trabalho.
- Maior incerteza sobre os requisitos significa mais atenção à modelagem de negócio.

Qualquer que seja o nível de detalhe e esforço que será colocado na modelagem de negócio, todo projeto deve iniciar com uma documentação preliminar que permita entender seu escopo. Um projeto com escopo mal definido certamente gerará desconforto ou mesmo desentendimentos entre o cliente e a equipe de desenvolvimento em função da inclusão (ou falta de inclusão) de determinadas características.

2.2 Visão geral do sistema

As atividades relacionadas à definição do escopo de um projeto devem usualmente tomar uma fração pequena do tempo total do projeto, embora variações possam existir dependendo do tipo do projeto: projetos com requisitos simples e bem definidos podem requerer não mais do que poucos dias de modelagem de negócio, enquanto projetos grandes e complexos podem precisar de semanas ou meses. No momento da definição do escopo, toda informação sobre o negócio da companhia deve ser explorada por meio de entrevista de usuários, clientes e especialistas de domínio, bem como pelo exame de documentos, relatórios, sistemas existentes e bibliografia relacionada.

Para a maioria dos projetos, a primeira questão a que um analista deve responder é a seguinte: qual é a visão da companhia para o projeto? Em outras palavras, o que a empresa quer com o projeto? Por que ele está sendo proposto e por que a empresa vai gastar dinheiro com ele?

Neste momento, outra questão, frequentemente esquecida, pode ser levantada: comprar ou desenvolver? Algumas vezes, o produto que o cliente quer está disponível para compra.

Essas questões devem ser respondidas em um tempo relativamente curto. Por isso sugere-se que a fase de Concepção seja relativamente rápida. Por quê? Porque nesse momento o cliente e o desenvolvedor usualmente ainda não têm uma ideia da real extensão do projeto, o que é, do ponto de vista de ambos, um investimento no futuro e, portanto, um risco.

Uma visão geral do sistema ou sumário executivo é um documento de formato livre no qual o analista pode relatar itens relevantes que foram descobertos sobre o sistema após as entrevistas iniciais com os interessados. O documento frequentemente inclui uma declaração de escopo para o projeto.

Não se pretende aqui definir regras para escrever este documento. Mas sugere-se que ele não seja muito longo. Algumas poucas páginas de texto e alguns diagramas costumam ser suficientes para descrever de forma resumida o escopo da maioria dos sistemas. Mais do que isso, poderia significar que muito detalhe foi incluído no sumário. Esses detalhes podem ser alternativamente abordados em outros documentos, como a lista de riscos, requisitos ou casos de uso.

A declaração de escopo deve apresentar, em linhas gerais, o produto que vai ser desenvolvido: o que vai ser incluído e o que *poderia* ser incluído, mas não vai. Essa informação pode ser obtida inicialmente entrevistando-se os interessados, e pode ser refinada depois com o uso dos diagramas apresentados neste capítulo.

Se possível, os principais entregáveis do projeto devem ser também definidos na visão geral, bem como os prazos em que o cliente pode esperar receber algum tipo de entregável da equipe de desenvolvimento. Normalmente, esta lista de entregáveis consiste em versões implementadas do software, mas a lista pode também incluir outros itens como design, manuais, mídia de instalação, treinamento etc. Pode ser muito difícil estabelecer prazos para entregáveis antes de fazer a análise de requisitos do sistema, porque as técnicas de estimação de esforço como *pontos de função* (Albrecht; Gaffney Jr., 1983), *pontos de caso de uso* (Karner, 1993) ou *COCOMO II* (Boehm, 2000) só podem ser aplicadas depois que uma boa parte dos requisitos é conhecida. Assim, a informação sobre prazos tão cedo nesta fase é mais uma tentativa do que um compromisso firme para desenvolvimento.

O documento de visão geral também pode conter alguns critérios de aceitação para o produto, ou seja, itens quantificáveis que possam ser usados para decidir se o projeto foi um sucesso ou não. Os critérios de aceitação do produto deveriam pelo menos incluir métricas para prazos, orçamentos e qualidade. Se critérios subjetivos como “cliente satisfeito”, “sistema fácil de usar” ou “tecnologia do estado da arte” forem utilizados, eles devem ser quantificáveis, ou seja, deve-se definir como medir a “satisfação do usuário”, “facilidade de uso” ou “estado da arte”, e assim por diante. Exemplos de critérios de aceitação quantificáveis são: “o sistema deve suportar até 50 mil acessos simultâneos sem degradar sua performance”; “o sistema deve eliminar a necessidade de usar papel para realizar os processos x e y ”; e “o sistema deve permitir vender até 100 livros por minuto pela internet”.

Mais informação pode ser adicionada ao documento de visão geral se necessário (por exemplo, principais riscos, tecnologias a serem usadas etc.). A visão geral do sistema é o documento de acordo básico para o cliente e o desenvolvedor. Ele será usado como base para o planejamento do resto do projeto.

É importante mencionar que, quando a visão geral está sendo construída, a análise de requisitos ainda não foi completada e, portanto, a informação mencionada na visão geral consiste em entendimentos preliminares. Espera-se que a análise de requisitos vá detalhar o escopo em profundidade, mas não aumentar sua extensão. Por exemplo, na análise de requisitos, o processo de vender livros pode ser detalhado com descontos, pedido, entrega, e assim por diante (considerando que “vender livros” tenha sido mencionado na declaração de escopo), mas se a folha de pagamento não foi mencionada na declaração de escopo, então uma inclusão deste item no projeto pode requerer uma renegociação da declaração de escopo.

Pressman (2008) propõe várias técnicas para comunicação com os interessados para a descoberta de objetivos de negócio, como, por exemplo:

- *Grupos de discussão tradicionais*, quando um analista treinado se encontra com um grupo de usuários finais ou pessoas que representam seus papéis.
- *Grupos de discussão eletrônica*, quando os encontros são conduzidos com o uso de mídia eletrônica.

- *Pesquisas iterativas*, quando várias pesquisas são realizadas entre os usuários finais, apresentando-se a eles algumas questões e solicitando esclarecimento.
- *Pesquisas exploratórias*, quando pesquisa exploratória é conduzida entre os usuários finais que usam um produto similar.
- *Construção de cenário*, quando um grupo de usuários finais é levado a produzir uma descrição de vários cenários de uso de um sistema.

A última opção parece ser bastante popular entre os praticantes de métodos ágeis, para os quais cenários são algumas vezes chamados de *histórias de usuário*.

A Figura 2.1 apresenta uma visão geral com uma declaração de escopo ainda *muito* preliminar para uma livraria virtual chamada *Livir*³, a qual será usada para ilustrar as técnicas de modelagem deste livro. Este texto poderia ter sido obtido após um rápido encontro preliminar com o cliente. Usando isso como uma base, um estudo mais profundo pode ser conduzido com casos de uso de negócio e outros diagramas, de forma que a informação contida no documento seja refinada, como mostrado adiante neste capítulo. Este documento é dividido em duas partes:

- Uma declaração de visão simples sobre o que o projeto vai oferecer aos clientes e ao negócio que está sendo criado.
- As restrições para a versão 1.0.

Visão Geral para o Projeto Livir

Versão 1.0

Visão:

Um novo negócio vai ser iniciado: uma livraria virtual. O sistema que vai dar suporte a ele deve gerenciar os processos de aquisição e venda da empresa. O acesso para clientes e gerentes deve se dar por um *website*.

Quando um livro é solicitado, ele é entregue imediatamente se estiver disponível em estoque; senão, o livro é solicitado a uma editora e um prazo compatível é informado ao cliente. O sistema deve calcular os impostos e a taxa de entrega, bem como aplicar descontos à venda se for o caso.

O sistema deve permitir que um gerente produza relatórios sobre os livros mais vendidos, sobre os clientes mais lucrativos, bem como sobre sugestões de livros baseados nos interesses passados do cliente.

Restrições:

- Clientes só podem pagar com cartão de crédito.
- A livraria vai trabalhar apenas com livros novos, não com usados.
- O acesso ao sistema só será possível por um *website*.

Figura 2.1

Visão geral do projeto Livir.

³ Por que uma livraria virtual? Porque é um típico sistema de informação comercial e a maioria dos leitores já deve ter experiência com este tipo de sistema. Além disso, porque ele é suficientemente simples, embora rico, para ilustrar as técnicas de modelagem deste livro. Tais sistemas já se encontram disponíveis para aquisição e personalização e possivelmente não faria mais sentido desenvolvê-los a partir do zero, mas o propósito aqui é puramente didático.

Para permitir a apresentação deste projeto em um livro, uma série de simplificações foi considerada. A descrição e a modelagem de um projeto real poderiam ser muito mais extensas. Além disso, os modelos apresentados podem, algumas vezes, ser parciais em um dado momento, sendo refinados ou modificados posteriormente.

Pode-se ver que a visão geral de um projeto é não estruturada. Ela inclui informação de negócio para gerentes e informação operacional para desenvolvedores. Algumas vezes, mesmo detalhes sobre tecnologia a ser usada podem ser descritos, se necessário. O que o analista deve considerar é que a visão geral descreve as principais preocupações dos clientes e que essas preocupações serão mais bem estruturadas mais tarde, durante a análise e o design.

Um bom analista deveria agir como um psicanalista, ou seja, deveria evitar falar no lugar do cliente. O analista deve ajudá-lo a falar; quando o cliente começar a relatar algumas de suas necessidades, o analista deve ouvir em vez de apresentar uma lista rapidamente (e, com frequência, equivocadamente) construída de soluções. O analista deve motivar o cliente a detalhar seus requisitos e questioná-los, de forma que o cliente possa decidir quais são urgentes e quais não são.

2.3 Casos de uso de negócio

Jacobson (1994) apresenta detalhadamente técnicas para modelagem de negócios com casos de uso de negócio. Em geral, em modelos de negócios, existem entidades (pessoas ou empresas) externas à organização-alvo, chamadas *atores de negócio*, e entidades internas chamadas *trabalhadores de negócio*. Casos de uso de negócio são os processos realizados por esses atores e trabalhadores que permitem que eles atinjam algumas das metas de negócio da organização.

O modelo de caso de uso de negócio considera a organização inteira como um sistema, e os atores podem ser pessoas, empresas ou outras organizações que criam ou mantêm relacionamentos de negócios com ela (Kroll; Kruchten, 2003).

A primeira coisa que a equipe deve pensar quando estiver fazendo a modelagem de negócio é que o que está sendo modelado *não é* um sistema de software; uma *empresa* está sendo modelada. A informação sobre o negócio é representada do ponto de vista de gerentes e especialistas de alto nível, já que eles conhecem e, muitas vezes, definem os objetivos da organização.

A quantidade de casos de uso de negócio que serão identificados pode ser relativamente pequena. Um processo de negócio é um processo de longo termo que é realizado pela empresa; pode-se pensar, por exemplo, em vender produtos, conduzir o marketing, fornecer serviços, resolver problemas dos consumidores etc. Cada processo deste tipo pode contribuir para um objetivo de negócio da empresa. Este não é o momento para detalhar tais processos. Por exemplo, vender produtos pode incluir muitos subprocessos, como registrar consumidores, oferecer produtos, enviar produtos, aplicar descontos. Mas esses processos (que mais tarde, possivelmente, serão identificados como casos de uso de

sistema, se forem automatizados) são envolvidos pelo processo de nível mais alto, que é *vender livros*.

Embora UML ainda não tenha um símbolo padrão para diferenciar casos de uso de negócio de casos de uso de sistema (Capítulo 3), é comum representar os casos de uso de negócio com o estereótipo <<business>>, ou cortando a elipse do caso de uso com uma aresta, como mostrado na Figura 2.2.

Neste exemplo, *Vender livros* é um caso de uso de negócio porque envolve uma relação entre a empresa e uma entidade externa (um comprador), produzindo um resultado perceptível e consistente: um ou mais livros vendidos. Este caso de uso possivelmente será realizado pelo comprador com a colaboração de um ou mais trabalhadores dentro da empresa; ele pode usar ou não sistemas de software.

Os nomes dos casos de uso devem ser escolhidos com cuidado, porque neste estágio eles vão resumir informações críticas sobre o sistema e escolhas erradas podem dificultar que os interessados tenham uma compreensão comum sobre a real intenção dos atores e trabalhadores de negócio. Blain⁴ apresenta algumas das melhores práticas coletadas de várias fontes:

- *Bons nomes de casos de uso refletem objetivos do usuário.* Expressões como *Acessar sistema* ou *Abrir janela principal* devem ser evitados, porque refletem tecnologia, e não um objetivo de negócio. Também nomes sem verbos, como *Pedido* devem ser evitados, porque não está claro o que vai acontecer com o pedido. Expressões melhores poderiam ser *Fazer pedido*, *Cancelar pedido*, *Pagar pedido* etc.
- *Bons nomes de casos de uso são os mais curtos possíveis.* Embora algumas pessoas possam propor que o nome de um caso de uso não deva ter mais do que duas ou três palavras (ou qualquer outro número), não é viável definir um limite como esse, porque existe o risco de perder a clareza sobre o objetivo de negócio se o nome for curto demais. Como Blain destaca, qual palavra poderia ser removida do nome *Receber pagamento atrasado* sem obscurecer seu significado? Entretanto, este nome é melhor do que *Receber pagamento atrasado de clientes que perderam a data do vencimento*.
- *Bons nomes de casos de uso têm verbos significativos.* Nomes de casos de uso não apenas devem ter verbos fortes, mas eles devem ser significativos. Um verbo não significativo é um que não deixa claro qual é o objetivo que o caso de uso realiza; por exemplo, o que um caso de uso como *Processar pedido* realiza? Que resultado ele produz? Já um nome como *Separar produtos para entrega* seria mais significativo neste caso.



Figura 2.2

Caso de uso de negócio.

⁴ Disponível em: <tynerblain.com/blog/2007/01/22/how-to-write-good-use-case-names/>.

- *Bons nomes de caso de uso têm voz ativa.* Como na escrita em geral, a voz ativa é preferível à voz passiva. *Pagar pedido* é melhor do que *O pedido é pago*.
- *Bons nomes de caso de uso estão no infinitivo.* O infinitivo indica o que o usuário está tentando fazer. Usar o tempo verbal no passado ou no futuro pode ser confuso e desnecessário. Por exemplo, *Pagar pedido* é melhor do que *Pagou pedido* ou *Pagará pedido*.
- *Bons nomes de casos de uso não identificam o ator.* Como os casos de uso são ligados aos atores, seus nomes não fazem parte do nome dos casos de uso. Assim, um caso de uso não deveria identificar o ator como em *Gerente cria relatório de vendas*. Deve-se apenas nomear o caso de uso como *Criar relatório de vendas* e ligá-lo ao gerente e outros atores, se necessário.
- *Bons nomes de casos de uso são consistentes.* As mesmas convenções de regra de nomeação devem ser aplicadas a todo o projeto ou grupo de projetos. Deve-se evitar, por exemplo, nomear um caso de uso como *Gerar relatório de vendas* e outro como *Produzir relatório de reservas*. Deve-se escolher um verbo para este significado e usá-lo consistentemente.

Adicionalmente, Probasco (2001) propõe que a lista de nomes de casos de uso deve se parecer com uma lista de coisas a fazer (*to do list*). Assim, em vez de escrever *Saque de dinheiro*, *Transferência de fundos* ou *Carregamento da máquina de saque*, seria melhor escrever *Sacar dinheiro*, *Transferir fundos* e *Carregar máquina de saque*.

Probasco também insiste que o significado de um nome de caso de uso deve ser preciso. Por exemplo, *Leilão* é um bom nome para um caso de uso? Não seria melhor usar um verbo como em *Executar leilão*? O verbo “executar” provavelmente seria a escolha de um programador, mas ele provavelmente não teria significado algum para um leiloeiro (o leilão vai ser colocado no paredão e fuzilado?). Neste caso, a ideia seria conversar com um especialista de domínio e obter um nome melhor. No entanto, mesmo o nome *Leiloar* não é suficientemente claro. Vai ser leiloado um único item ou um lote de itens? Ou ainda é simplesmente um participante do leilão dando um lance? Neste caso, nomes como *Leiloar lote de itens*, *Leiloar item* ou *Apresentar lance* seriam mais claros.

2.3.1 Atores de negócio e trabalhadores de negócio

Durante a modelagem de negócio há dois tipos de atores que podem ser referenciados (Figura 2.3):

- *Atores de negócio:* pessoas, organizações ou mesmo sistemas que executam algumas atividades pertencentes ao processo, mas que não são parte da empresa-alvo. Ou seja, eles não estão sob o controle da organização. Atores de negócio poderiam ser, por exemplo, clientes, fornecedores, auditores externos, empresas parceiras ou mesmo outros sistemas automatizados com os quais a empresa deve interagir.

- *Trabalhadores de negócio*⁵: pessoas, organizações ou mesmo sistemas que realizam atividades que pertencem ao processo e que são parte da empresa-alvo. Poderiam ser os funcionários da empresa, seus departamentos ou mesmo sistemas de software existentes que pertencem à empresa. Graficamente, eles são distinguidos dos atores de negócio pelo estereótipo <<worker>>.

Esta diferenciação é importante porque atores de negócio não podem ser automatizados, ou seja, eles não podem ser substituídos por sistemas computacionais. Entretanto, certos papéis de trabalhadores de negócio podem possivelmente ser substituídos por sistemas automáticos (English, 2007).

Atores de negócio e trabalhadores de negócio estão ligados aos casos de uso de negócio dos quais eles participam por linhas sem setas⁶, como mostrado na Figura 2.4.

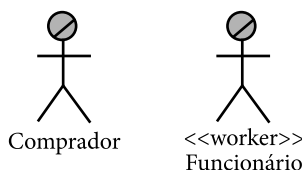


Figura 2.3

Ator de negócio (à esquerda) e trabalhador de negócio (à direita).

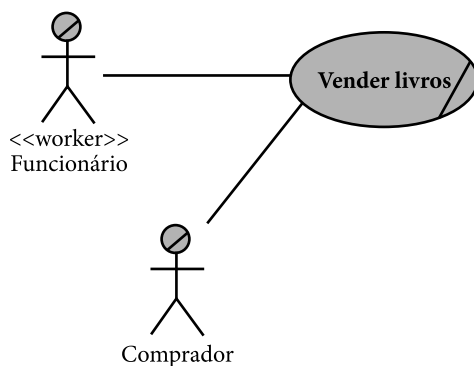


Figura 2.4

Ligações entre atores de negócio, trabalhadores de negócio e casos de uso de negócio.

⁵ Se estivermos modelando uma empresa como um todo, os trabalhadores de negócio usualmente não aparecem, pois eles são internos ao sistema da empresa, mas se conexões entre diferentes subsistemas da empresa são mostradas, então os trabalhadores de negócio devem aparecer.

⁶ Algumas abordagens sugerem o uso de setas para indicar se o ator apenas recebe ou envia informação para o sistema. Mas, na prática, além de não ser suportado pela UML 2, isso praticamente não traz informação útil neste ponto da análise de negócio, porque nesse momento o mais importante é descobrir quais casos de uso realmente existem e quais atores estão envolvidos com eles. Os detalhes da interação serão mostrados mais tarde com os diagramas de atividade, os quais descrevem como um caso de uso de negócio é realizado.

2.3.2 Oportunidades de automação

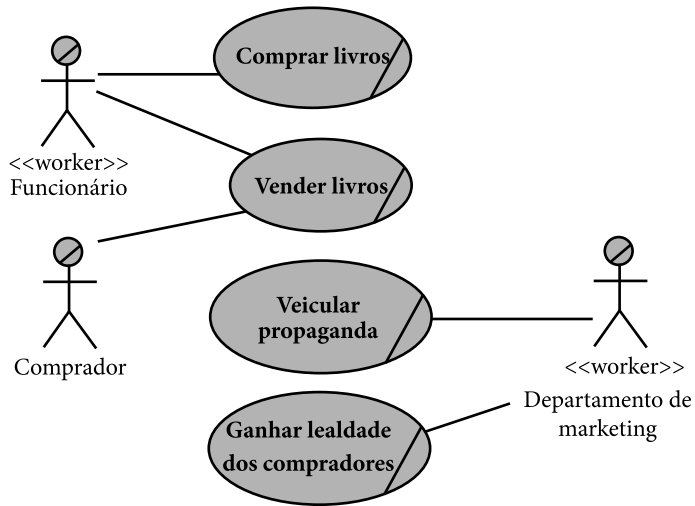
Uma livraria que pretenda vender pela internet, como Livir, deve produzir um modelo de negócio que identifique seus principais casos de uso de negócio, atores externos (compradores, editoras etc.) e trabalhadores de negócio (gerentes, funcionários etc.). Se supusermos que a livraria já existe, mas não possui nenhuma ou muito pouca automação, então os papéis dos trabalhadores de negócio serão executados basicamente por pessoas reais. As funções desses trabalhadores podem ser candidatas a automação se houver interesse em aumentar o nível de automação da empresa. Por exemplo, o papel do funcionário que atualmente ajuda os compradores a encontrar os livros nas prateleiras pode ser substituído na livraria virtual por um sistema de guia automatizado. Entretanto, o papel do comprador não pode ser substituído por um sistema automatizado.

Nem todo papel de trabalhador de negócio será usualmente substituído. Isso dependerá do escopo do projeto a ser definido. Uma das vantagens de fazer o diagrama de casos de uso de negócio, então, é ajudar na visualização e na tomada de decisão sobre automação. Com o diagrama, é possível visualizar o que deve ficar dentro e fora do escopo de automação, dependendo dos objetivos do projeto.

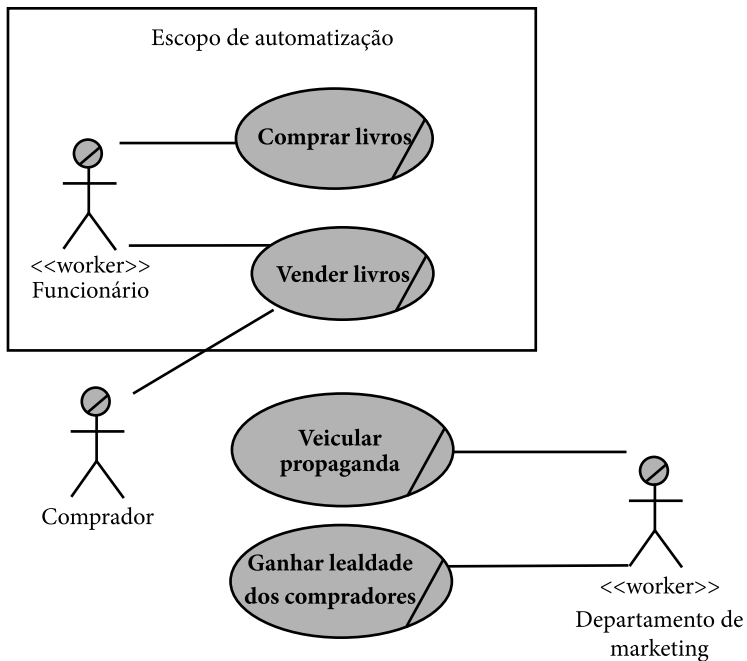
Normalmente, durante o processo de informatização de uma organização, pode ser útil substituir pelo menos os papéis de trabalhadores que correspondem a meros procuradores (*proxies*), ou seja, aqueles que simplesmente realizam ações no lugar de um ator externo. No caso da livraria, o papel do funcionário de vendas é necessário apenas para ajudar o cliente na loja física, mas quando as vendas são movidas para o espaço virtual, aquele papel já não é mais necessário, porque o cliente pode facilmente encontrar seu caminho no *website*.

A associação dos casos de uso de negócio com as principais metas de negócio da empresa ajuda a identificar o escopo do sistema, isto é, as atividades que serão efetivamente automatizadas no projeto que está sendo iniciado. No exemplo da Figura 2.5, todos os casos de uso são candidatos a automação com diferentes níveis de prioridade. Entretanto, a empresa pode estar interessada apenas em implementar compras e vendas neste momento, deixando o marketing e os serviços ao consumidor para um segundo projeto.

A fronteira do sistema pode ser usada para identificar o conjunto de casos de uso de negócio e os trabalhadores de negócio que serão automatizados, como mostrado na Figura 2.6.

**Figura 2.5**

Casos de uso de negócio: candidatos a automatização.

**Figura 2.6**

Fronteira de sistema sendo usada para indicar o escopo do projeto.

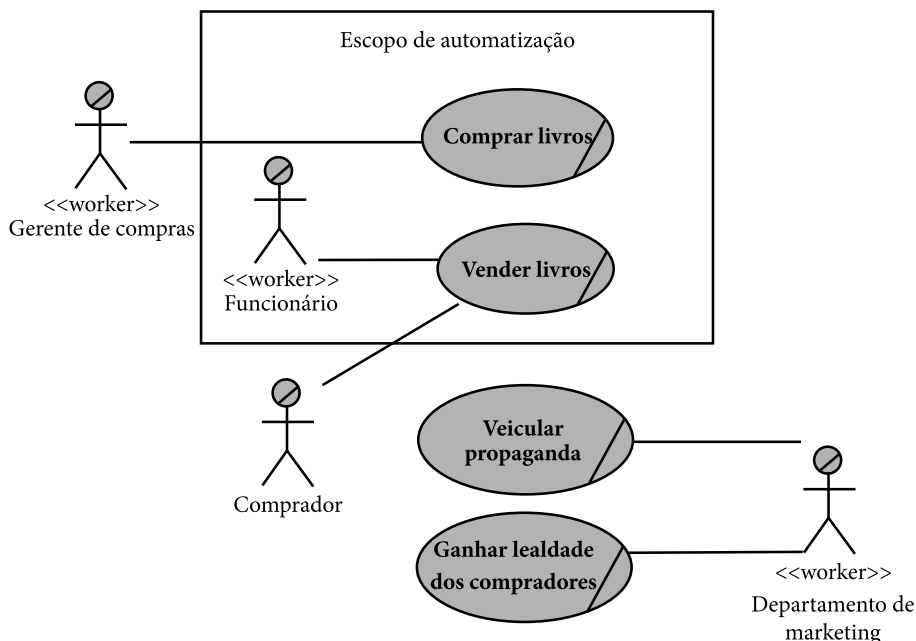


Figura 2.7

Um ator é dividido porque apenas parte de seu papel será automatizada.

O diagrama de casos de uso de negócio mostra que o projeto de automatização vai incluir os processos de comprar e vender livros e o papel do funcionário. Ele também mostra que os processos de veicular propaganda e ganhar lealdade do cliente estão fora do escopo do projeto.

Entretanto, uma análise mais profunda poderia mostrar que o funcionário que compra livros para a livraria não pode ser facilmente automatizado. Poderia até ser o caso de que os livros fossem comprados automaticamente por um sistema que poderia analisar vendas potenciais e estoque sem a participação de qualquer ser humano. Mas, se este não é o caso, a melhor coisa a fazer é considerar que o funcionário realmente desempenha dois papéis diferentes: ajudar o comprador (o papel que será automatizado) e comprar livros para a livraria (o papel que não será automatizado). A Figura 2.7 mostra uma evolução do diagrama onde parte das responsabilidades do funcionário é reatribuída a um trabalhador chamado “Gerente de compras”. Não existe uma equivalência “um para um” entre um papel e uma pessoa: uma pessoa pode desempenhar vários papéis e um papel pode ser desempenhado por várias pessoas diferentes.

Se o projeto, no entanto, consistir em vários subprojetos ou estágios, nos quais diferentes partes da empresa serão automatizadas, isso pode ser indicado pelo uso de diferentes fronteiras no diagrama: uma para cada subprojeto ou estágio.

O próximo passo na direção da análise de requisitos é o estudo detalhado dos casos de uso de negócio que serão automatizados. O nível de precisão e detalhe depende dos objetivos do projeto, conforme já foi discutido anteriormente neste capítulo.

2.4 Diagrama de atividade de negócio

O *diagrama de atividades* da UML é uma opção muito popular para estudar os detalhes de um caso de uso de negócio. Com este diagrama podem ser especificadas as diferentes atividades realizadas pelos atores e trabalhadores de negócio na direção da meta geral do caso de uso que está sendo informatizado. Os diagramas de atividades podem ser usados para representar processos no nível organizacional.

2.4.1 Elementos básicos

O diagrama de atividades da UML é em muitas coisas similar a um fluxograma. Existem atividades sequencialmente ligadas. Pontos de decisão e repetição podem ser usados e mesmo paralelismo pode ser expresso nesses diagramas.

O diagrama de atividades pode ser dividido em *raias*, cada uma representando um ator ou trabalhador que participa de um conjunto de *atividades*. Como no diagrama de casos de uso, um ator ou trabalhador pode ser um papel representado por uma pessoa, departamento, sistema computacional ou mesmo uma organização completa.

O processo representado no diagrama usualmente possui um *nodo inicial*, representado por um círculo sólido ●, e um *nodo de atividade final*, representado por um círculo sólido inscrito em uma circunferência ⊙. O nodo inicial pode não ser considerado obrigatório em alguns casos, quando o início do processo pode ser inferido pela observação de uma única atividade que não tenha um antecessor. Entretanto, o uso dessa marca melhora a legibilidade dos diagramas porque é mais fácil perceber onde o processo realmente inicia.

As atividades são representadas por *nodos de ação* . Quando uma atividade é representada dentro de uma raia, isso significa que o ator correspondente é o responsável pela atividade.

Fluxos ou *dependências* entre atividades são representadas por setas →. Fluxos usualmente ligam duas atividades, indicando precedência entre elas.

Um caminho é uma sequência de atividades ligadas por fluxos:  →  →  →  → .

No exemplo de Livir, o diagrama de atividade pode ser usado como uma forma de visualizar os detalhes de casos de uso de negócio, tais como *Vender livros*. O modelo apresentado na Figura 2.8 mostra uma primeira abordagem para este processo.

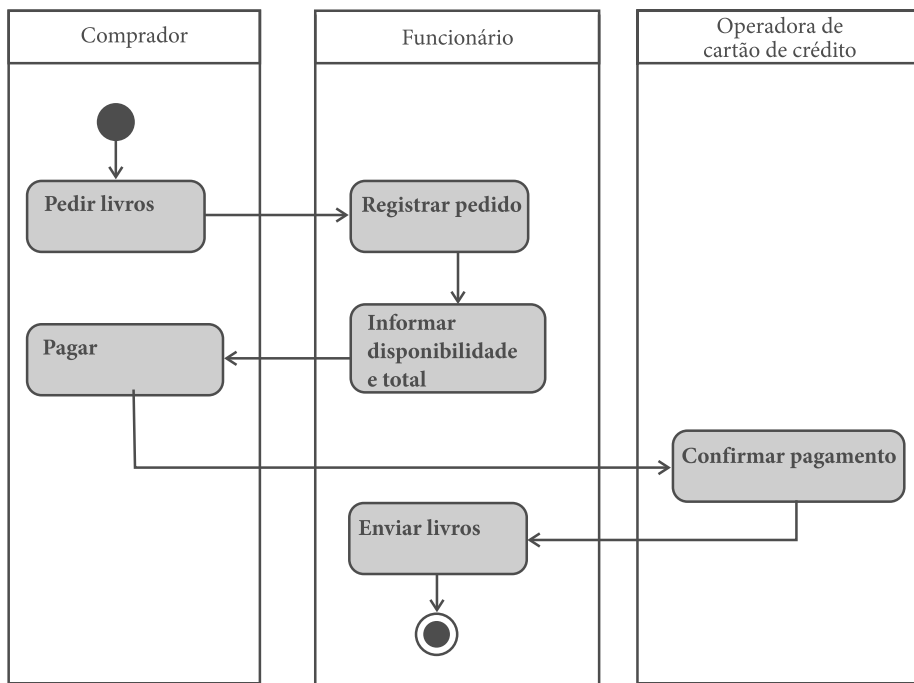


Figura 2.8

Primeira abordagem para um diagrama de atividades para modelar o processo de negócio de venda de livros.

O diagrama que começou a ser desenhado na Figura 2.8 não pretende ser um modelo para o sistema que vai ser construído e não deve ser pensado como tal. Sua utilidade consiste em ajudar a equipe a compreender quais atividades e atores estão envolvidos com os principais processos de negócio da empresa, tal que, com o uso desta informação, ela possa realizar uma análise de requisitos mais efetiva posteriormente. O caminho lógico do diagrama deve ser entendido e validado pelos especialistas no negócio.

Mais tarde, os analistas vão examinar em detalhe cada uma das atividades que devem ser realizadas. Se uma dada atividade pertence ao escopo do sistema que vai ser implementado, então ela será alvo de uma análise mais detalhada.

2.4.2 Nodos de controle de fluxo

Dois tipos de nodos de controle de fluxo são comuns nos diagramas de atividade – nodos de decisão e nodos de paralelismo:

- Os *nodos de decisão* (*branch e merge*) são representados por losangos . Os fluxos que saem de um nodo de *branch* devem ter condições de guarda (uma expressão lógica entre colchetes). Dois ou mais fluxos podem deixar um nodo *branch*, mas é importante que as condições de guarda sejam mutuamente exclusivas, ou seja, apenas uma delas pode ser verdadeira de cada vez.

- Os *nodos de paralelismo* (*fork* e *join*) são representados por barras . Os fluxos que saem de um nodo *fork* são executados em paralelo.

O diagrama da Figura 2.8 é ainda uma representação muito grosseira do processo real de venda de livros. Apenas para ilustrar uma possível evolução do diagrama, pode-se examinar a situação na qual alguns livros pedidos não estão disponíveis em estoque. Seria necessário pedi-los de um dos fornecedores e adicioná-los ao pedido do comprador após sua chegada. A Figura 2.9 mostra esta situação indicando que, se alguns livros não estão disponíveis, então eles devem ser solicitados aos fornecedores, e o pedido de compra é enviado apenas depois da chegada destes livros.

Logo abaixo da atividade *Confirmar pagamento* existe um nodo *branch* representado por um losango. Como mencionado antes, este nodo de decisão permite que apenas um dos fluxos de saída seja seguido. As condições de guarda (*[alguns livros faltando]* e *[todos os livros em estoque]*) são mutuamente exclusivas. A partir do nodo *branch*, o fluxo segue um dos caminhos alternativos até atingir o nodo *merge*, o qual aparece logo abaixo da atividade *Enviar livros*.

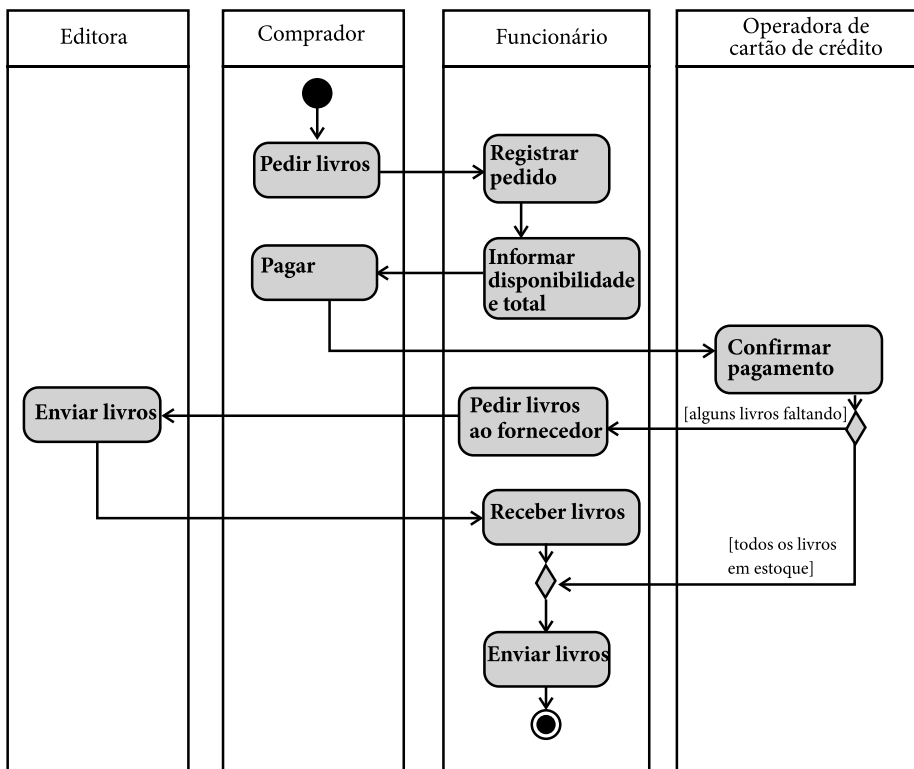
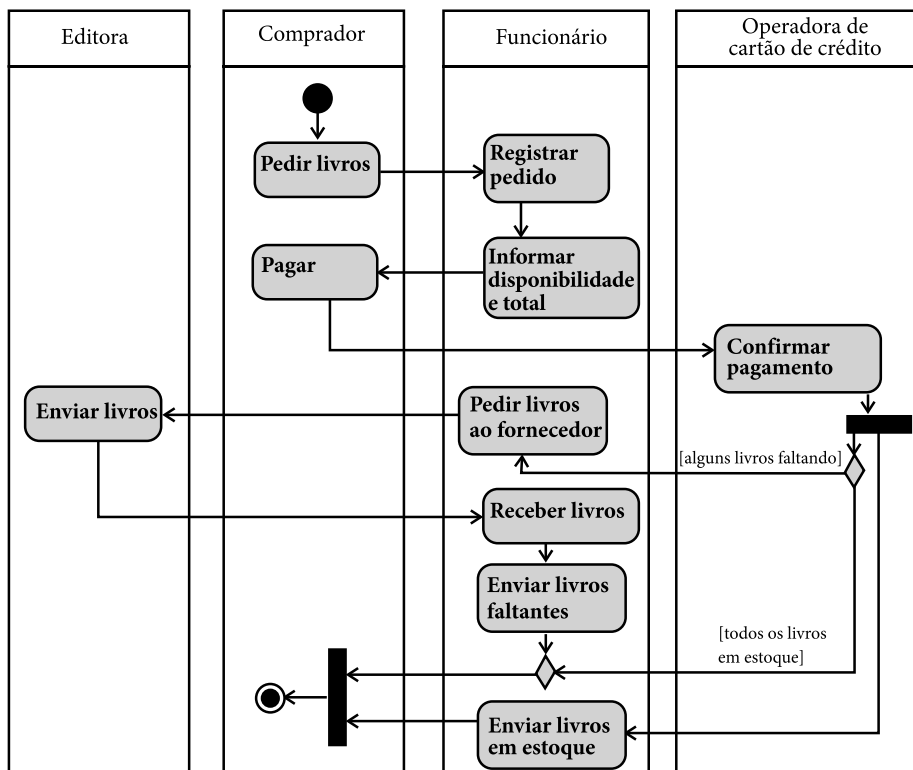


Figura 2.9

Exemplo de fluxo de controle de decisão.


Figura 2.10

Exemplo de fluxos paralelos.

Mais tarde, o analista poderá descobrir que o modelo ainda não é satisfatório. Por exemplo, mesmo se alguns livros não estiverem em estoque, o pedido poderia ser enviado em duas ou mais entregas. Assim, dois caminhos seriam realizados em paralelo: enviar os livros que estão disponíveis em estoque e, se necessário, pedir os outros livros e enviá-los depois, como mostrado na Figura 2.10.

A barra abaixo da atividade *Confirmar pagamento*, na Figura 2.10, representa um nodo *fork*, porque ela inicia dois caminhos paralelos, e a outra barra representa o nodo *join* porque sincroniza os caminhos paralelos em um único caminho.

O caminho único após o nodo *join* só pode ser seguido quando todos os caminhos paralelos que chegam ao nodo *join* tenham sido seguidos. Pode ser visto no modelo que se todos os livros estiverem no estoque apenas um dos caminhos paralelos terá atividades para serem realizadas, já que, neste caso, o outro caminho irá imediatamente para os nodos *merge* e *join*.

Os nodos *fork*, *join*, *branch* e *merge*, bem como os nodos inicial e final, podem ser colocados dentro de raias. Entretanto, isso não afeta sua semântica. Assim, a escolha sobre o local onde colocar cada um desses nodos é apenas estética.

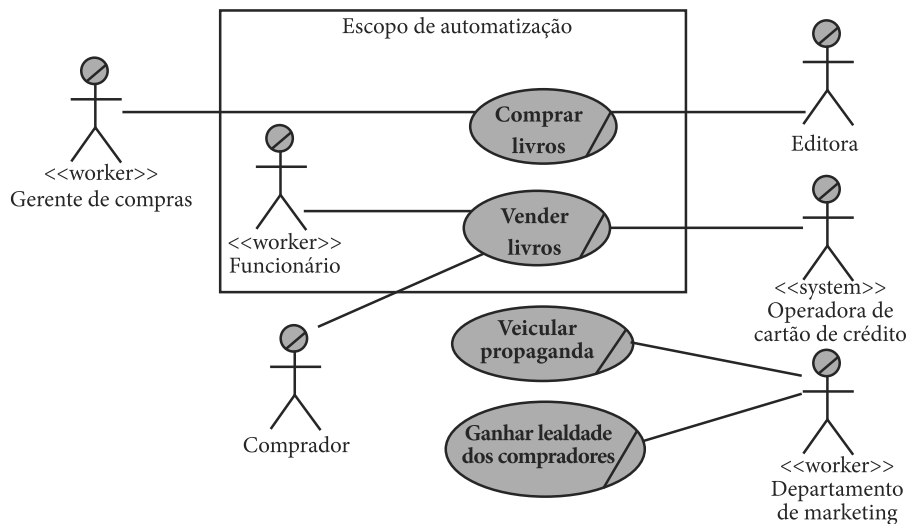


Figura 2.11

Diagrama de casos de uso de negócio atualizado com informação descoberta com o diagrama de atividades.

Outro nodo que pode ser útil algumas vezes é o *nodo de final de fluxo* $\otimes$, o qual termina um caminho (paralelo ou não) que esteja sendo executado. A diferença entre ele e o nodo de atividade final é que, no caso do nodo de atividade final, todos os fluxos do diagrama são terminados quando um único fluxo o atinge, enquanto no caso do nodo de final de fluxo apenas um único fluxo (entre outros paralelos) é terminado, mas a atividade continua.

No caso da Figura 2.7, apenas um ator (o cliente) e um trabalhador (funcionário) participam do caso de uso *Vender livros*. Entretanto, após detalhar as atividades relacionadas àquele caso de uso, como feito na Figura 2.10, foi descoberto que mais dois atores eram necessários: a *Editora* e a *Operadora de cartão de crédito*. Adicionalmente, a *Editora* é um ator que deve ser ligado ao caso de uso *Comprar livros*. Assim, dependendo do nível de detalhe desejado neste ponto do projeto, o diagrama de caso de uso de negócio pode ser atualizado para refletir essas descobertas, como mostra a Figura 2.11.

Outras opções para detalhar um diagrama de casos de uso de negócio são os diagramas de sequência e de comunicação da UML. Entretanto, esses diagramas são usados para representar envio de mensagens entre seus elementos. Nem sempre é natural encontrar nomes para rotular essas mensagens no caso de um processo de negócio, e nomes genéricos como “enviar dados” e “verificar resultado” acabam sendo escolhidos no final. Assim, o diagrama de atividades é a escolha mais natural para descrever o que acontece no mundo real, dentro de uma organização de pessoas. Entretanto, como visto na Seção 5.8, diagramas de sequência são muito úteis para detalhar casos de uso de sistema, porque estes usualmente consistem em uma sequência de fluxos de informação sendo trocada entre os atores e o sistema.

2.5 Aspectos de negócio dependentes de estado

Além dos casos de uso de negócio, que podem ser detalhados por diagramas de atividades, pode ser importante entender outros aspectos de um negócio que nem sempre aparecem claramente nesses diagramas, como alguns objetos-chave de negócio que mudam de estado durante seu ciclo de vida.

Assim, outro diagrama UML que ajuda a equipe a entender o negócio da organização é o *diagrama de máquina de estados*. Assim como o diagrama de atividades, trata-se de diagrama comportamental, mas, em vez de modelar atividades em processos, ele representa um conjunto de estados nos quais um sistema, ator ou entidade pode estar em um determinado instante.

O diagrama de máquina de estados especifica *eventos* em seus fluxos. Um evento dispara um fluxo que leva a entidade de um estado para outro. Ou seja, os fluxos são rotulados com eventos que, se ocorrerem, fazem a entidade mudar de um estado para outro.

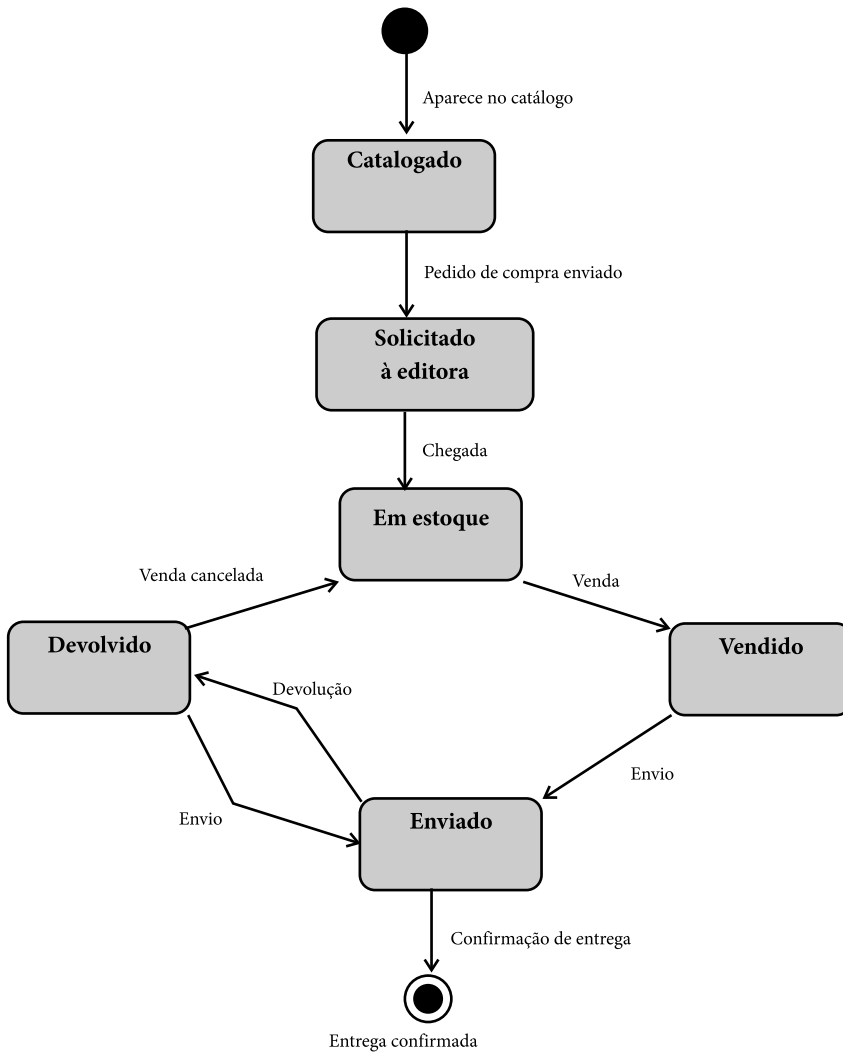
Essas ocorrências podem ser, entretanto, restritas por *condições de guarda*. Por exemplo, o evento *login* só poderia mudar o sistema para o estado *logado* se a senha estiver correta.

Graficamente, os eventos são representados nos fluxos por expressões simples e as condições de guarda são representadas entre colchetes, como no caso dos diagramas de atividade.

As transições em um diagrama de máquina de estados também podem incluir *efeitos* que ocorrem em resposta a uma transição de estado. Por exemplo, a transição ativada pelo evento *login* e guardada pela condição [*senha correta*] pode ativar a ação */liberar acesso*, que é um efeito da transição (não sua causa). Ações associadas às transições são representadas por expressões iniciadas por uma barra (/).

Como um exemplo de como este diagrama pode ser útil durante a análise do negócio, pode-se imaginar que uma equipe está interessada em entender o ciclo de vida de um livro em uma livraria; examinar os diferentes estados de um objeto-chave de negócio tal como um livro para este projeto pode reduzir o risco de subestimar a complexidade dos requisitos do sistema.

Assim, a equipe pode descobrir que o livro é inicialmente registrado em um catálogo por uma editora. A livraria pode então pedir uma quantidade de cópias do livro. Quando a encomenda chega, o livro é adicionado ao estoque e torna-se disponível para venda. Uma vez vendido, o livro é removido do estoque e enviado ao departamento de remessa. Em algumas situações, livros podem ser devolvidos à livraria, por exemplo, em razão de um endereço incorreto, ou por terem sido danificados durante o envio. No primeiro caso, a livraria deve contatar o comprador e, dependendo do contato, ela pode cancelar ou reenviar o livro. O ciclo de vida do livro termina apenas quando a empresa de transporte confirma a entrega da encomenda. Todas essas mudanças de estado (transições) são representadas na Figura 2.12.

**Figura 2.12**

Primeiro modelo de máquina de estados para o ciclo de vida de um livro.

Entretanto, o modelo apresentado na Figura 2.12 é ainda incompleto em relação a possíveis transições de um livro. Por exemplo, um livro enviado pode ser devolvido danificado e, neste caso, não pode ser reenviado. Isso pode ser representado pela adição de uma condição de guarda à transição que vai do estado *Devolvido* para o estado *Enviado*, e pela adição de um novo estado final no qual o livro é descartado, como mostrado na Figura 2.13.

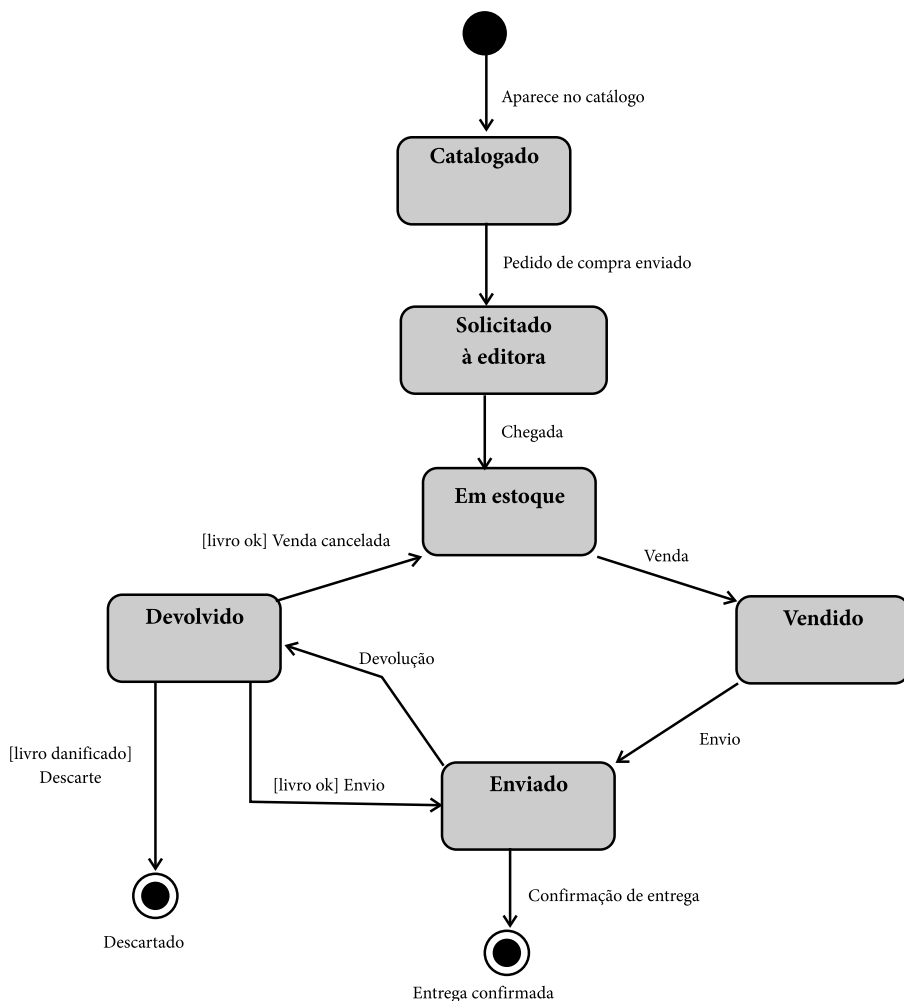

Figura 2.13

Diagrama de máquina de estados com condições de guarda.

Essas condições de guarda são ainda bastante informais. Isso é aceitável durante a fase de modelagem de negócio, porque o seu objetivo é entender a organização do negócio, e não a especificação de um sistema computacional. Apesar disso, condições de guarda mais precisas podem ser obtidas se uma linguagem formal como *Object Constraint Language* (OCL) for usada. Mais adiante, neste livro, expressões utilizando OCL serão introduzidas.

Em algumas situações, é possível que um evento cause uma transição a partir de mais do que um único estado. No exemplo da Figura 2.13, é possível imaginar que um livro pode ser danificado não apenas no estado *Devolvido*, mas também quando ele estiver em

outros estados como *Em estoque* e *Vendido*. Então, deveria existir uma transição para *Descartado* a partir desses três estados. Entretanto, para abreviar este conjunto de transições, um *superestado* pode ser usado.

Na Figura 2.14, qualquer transição do superestado *No depósito* representa um conjunto de transições para cada um de seus subestados. No caso de uma transição *para* o superestado, é necessário estabelecer qual dos subestados será o inicial. Para fazer isso, pode-se incluir um nodo inicial dentro do superestado ou, em vez disso, fazer cada transição ir diretamente para um subestado.

Um objeto pode estar simultaneamente em mais do que um único estado. Por exemplo, um livro pode estar ou não em oferta especial desde o momento em que for solicitado ao fornecedor até o momento em que ele atinja um estado final como *Descartado* ou *Entregue*. Assim, um livro pode ser modelado com dois estados concorrentes: o primeiro estado determina se ele é ou não uma oferta especial e o segundo estado determina seu *status* em relação à venda. No diagrama da Figura 2.15, os estados concorrentes são representados dentro de *regiões ortogonais* em um superestado.

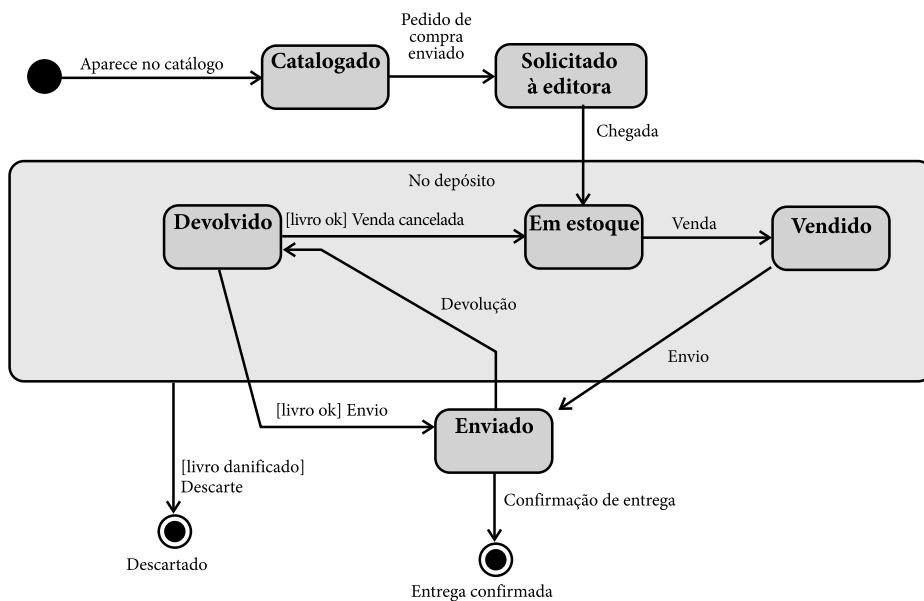


Figura 2.14

Diagrama de máquina de estados com um superestado.

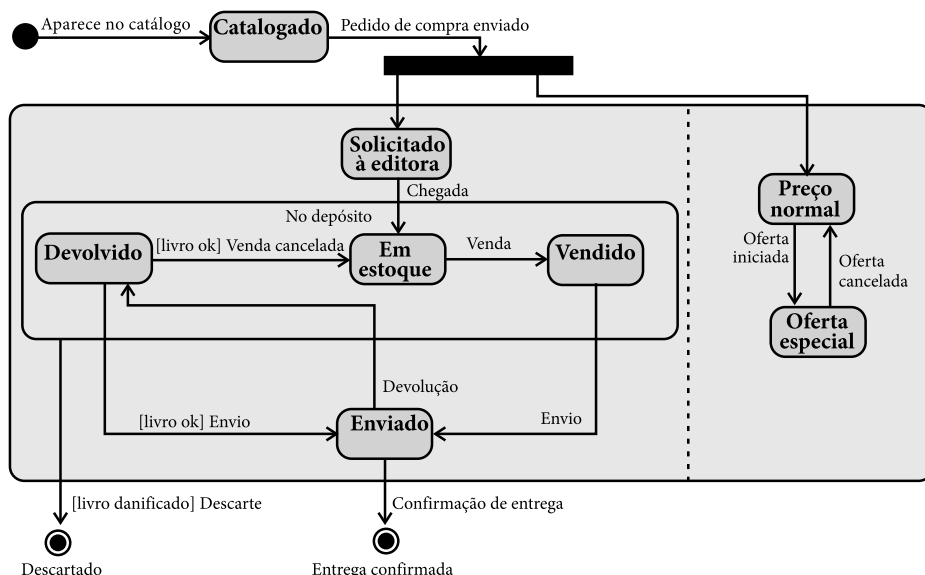

Figura 2.15

Diagrama de máquina de estados com regiões.

Quando uma transição deixa um superestado, então todos os subestados concorrentes também são deixados. Por exemplo, quando um evento *Confirmação de entrega* ou *Descarte* ocorre na Figura 2.15, ambos os estados concorrentes deixam de estar ativos e o livro é considerado em estado final.

A análise dos estados de um livro poderia continuar. Por exemplo, seria possível determinar que um livro nos estados *Vendido*, *Devolvido* ou *Enviado* não poderia trocar seu preço de *Preço normal* para *Oferta especial* ou vice-versa. Isso poderia ser modelado pela adição de uma condição de guarda às transições dos estados *Preço normal* e *Oferta especial*: *[não está nos estados Vendido, Devolvido ou Enviado]*.

O uso do diagrama de máquina de estados para representar objetos de negócio é útil para aprimorar a descoberta de requisitos de sistema relacionados com alguns objetos-chave do negócio, tal como será mostrado na Seção 3.4.

2.6 Comentários

É importante lembrar que o objetivo da modelagem de negócio é usualmente vislumbrar uma visão geral do negócio, e não efetuar uma especificação detalhada de seus processos. Assim, na maioria dos casos, o objetivo da equipe é concentrar-se na descoberta de informações sobre o negócio, e não na especificação formal do funcionamento do negócio como se este fosse uma máquina.

Uma questão deve ser colocada: quais elementos de negócio merecem a produção de um diagrama de máquina de estados ou de atividade? Salvo melhor juízo, não é recomendável preparar um diagrama para qualquer elemento de negócio porque, se isso for feito, a fase de Concepção poderá levar muito tempo para ser concluída e sua objetividade será prejudicada. O que é usualmente necessário neste ponto é um modelo para alguns elementos-chave, de forma que o comportamento seja mais bem compreendido e que seus requisitos possam ser derivados depois.

Uma pista para identificar esses elementos-chave é perguntar quais são os objetos de negócio. No caso de uma livraria, são os livros; no caso de um hotel, as hospedagens; no caso de um tribunal, os processos criminais, e assim por diante. Assim, esses são os elementos-chave que devem ser modelados em maior detalhe para ajudar a entender o negócio.

Uma segunda questão poderia ser a seguinte: qual diagrama deve ser usado – atividade ou máquina de estados? A resposta depende da natureza do elemento a ser modelado. Um estado não corresponde necessariamente a uma atividade. Uma TV, por exemplo, pode estar no estado *desligada* quando não está fazendo nada. Um diagrama de atividades é útil quando está modelando pessoas, organizações ou sistemas fazendo coisas. No entanto, um diagrama de máquina de estados é útil quando uma única entidade passa por diferentes estados nos quais ela não está necessariamente fazendo coisas. Além disso, o diagrama de atividades usualmente detalha um caso de uso de negócio (ou seja, um processo como vender, comprar, verificar etc.), enquanto o diagrama de máquina de estados usualmente está associado a um objeto de negócio (como livros, pessoas, pedidos etc.).

A modelagem de negócio não consiste, entretanto, apenas em construir diagramas. O propósito dos diagramas é ajudar a entender o contexto e os requisitos iniciais de um projeto e um sistema. Modelagem de negócio é uma das atividades-chave que ajudam a equipe a identificar e se preparar para riscos de projeto. Outras atividades de mitigação de riscos podem também ser realizadas durante a Concepção, tais como prova de conceito, *workshops*, prototipação, testes preliminares etc. Qualquer estratégia que ajude a entender o quadro mais amplo e identificar os maiores riscos e complexidades de um projeto é válida.

2.7 O processo visto aqui

	Concepção	Elaboração
Modelagem de negócio	Construir uma visão geral do sistema, incluindo, na medida do necessário: • um diagrama de casos de uso de negócio e determinar o escopo de automação para o projeto • diagramas de atividades de negócio preliminar para os casos de uso de negócio • diagramas de máquina de estados para objetos-chave de negócio	
Requisitos		
Análise e design		
Implementação		
Teste		
Gerenciamento de projeto		

2.8 Questões

1. Qual é o propósito da modelagem de negócio?
2. A modelagem de negócio pode ocorrer com diferentes níveis de intensidade para diferentes tipos de projeto. Explique e apresente exemplos.
3. Qual a diferença entre um ator de negócio e um trabalhador de negócio?
4. Que aspectos de um negócio podem ser detalhados por um diagrama de atividades? Quais podem ser detalhados por um diagrama de máquina de estados?

Requisitos de alto nível

TÓPICOS-CHAVE NESTE CAPÍTULO

- Atores de sistema
 - Casos de uso de sistema
 - Como encontrar casos de uso de sistema no modelo de negócio
 - Requisitos
 - Modelo conceitual preliminar
-

3.1 Introdução aos requisitos de alto nível

Uma vez que o negócio tenha sido razoavelmente entendido e modelado, a análise de requisitos pode começar. Embora existam várias abordagens para representar requisitos, este livro apresenta uma técnica baseada em casos de uso de sistema.

O *caso de uso de sistema* é um processo individual que é identificado a partir das atividades de sistema (Seção 2.4). Casos de uso de sistema de *alto nível* (ou *resumidos*) representam os *requisitos funcionais* de mais alto nível de um sistema. As *anotações* nesses casos de uso representam os *requisitos não funcionais*, ou seja, as restrições e as qualidades relacionadas àquelas funções. Os *requisitos suplementares* são requisitos não funcionais que se aplicam ao sistema como um todo – não apenas às funções individuais. Eles podem ser representados no documento de *especificações suplementares* (Seção 3.5.8).

Casos de uso de sistema são úteis para muitas atividades relacionadas ao desenvolvimento de sistemas, como:

- *Definição e validação da arquitetura do sistema*: em geral, classes, associações e atributos que formam parte da arquitetura do sistema são obtidos a partir dos textos dos casos de uso¹ (Seção 3.6).
- *Criação de casos de teste*: casos de uso podem ser vistos como uma base para os testes de sistema e de aceitação (Seção 11.7), nos quais a funcionalidade é verificada do ponto de vista do usuário.
- *Planejamento de iterações*: cada caso de uso recebe uma prioridade (Seção 4.3), e o esforço para desenvolvê-lo é estimado (Seção 4.2), de forma que a equipe possa decidir quais casos de uso desenvolver a cada iteração.
- *Base para documentação do usuário*: casos de uso são descrições dos fluxos de operação normal do sistema, bem como dos fluxos alternativos que representam a forma de

¹ Daqui por diante, casos de uso de sistema serão referenciados simplesmente como *casos de uso*, exceto quando forem comparados com casos de uso de negócio.

lidar com condições excepcionais eventuais (Capítulo 5). Essas descrições são uma excelente base para iniciar o manual do usuário, porque todas as possíveis funcionalidades estão descritas ali de forma estruturada e completa.

Cada caso de uso representa um conjunto coerente de requisitos funcionais de sistema. Usualmente, mais do que uma função é relacionada a cada caso de uso, especialmente se este for complexo. Algumas funções, no entanto, podem ser associadas a mais do que um caso de uso. Em algumas situações, pode também acontecer que uma função corresponda a um único caso de uso e vice-versa. Isso usualmente ocorre com casos de uso muito simples, tais como relatórios ou gerenciamento de entidades.

Há pelo menos três abordagens que relacionam requisitos e casos de uso:

- Criar uma lista de requisitos funcionais e, então, identificar os casos de uso a partir dela.
- Criar uma lista de casos de uso e, então, extrair os requisitos funcionais dela.
- Considerar que casos de uso *são* requisitos. A versão em alto nível dos casos de uso corresponde aos requisitos de alto nível e a versão expandida dos casos de uso corresponde ao conjunto completo de requisitos.

Neste livro, a terceira abordagem é escolhida, porque ela é mais simples, direta e evita a geração de diferentes documentos com o mesmo objetivo. Eliciação e análise de requisitos correspondem então à descoberta de casos de uso de sistema e suas propriedades.

3.2 Atores de sistema

Um *ator de sistema* é uma entidade do mundo real que interage com o sistema por meio de um caso de uso. Atores podem ser papéis representados por pessoas tais como compradores, fornecedores, vendedores, operadores etc. Atores também podem ser sistemas externos, ou seja, sistemas que estão fora do escopo do projeto sendo desenvolvido.

Atores humanos e sistemas externos interagem com o sistema-alvo pelo envio e recebimento de informação por meio de uma interface. No caso de atores humanos, a informação é usualmente trocada por dispositivos de entrada, como teclado, mouse ou outros especiais. Estes atores recebem informação do sistema por meio de interfaces, como telas, impressoras ou outros dispositivos especiais.

A comunicação com atores que são sistemas externos usualmente acontece por meio de uma rede de computadores. Nesse caso, a interface de comunicação consiste em uma rede e seus protocolos.

A ideia de sistemas externos como atores não deve ser confundida com a dos subsistemas internos que são componentes do sistema em desenvolvimento. Por exemplo, um sistema gerenciador de banco de dados (DBMS) usado para implementar a persistência de dados para o sistema em desenvolvimento não é um ator, mas um componente da arquitetura interna do sistema. As seguintes regras podem ajudar a identificar apropriadamente os sistemas externos que poderiam ser atores:

- Sistemas atores são *sistemas de informação completos*, e não apenas bibliotecas de classes ou componentes. Esses sistemas devem armazenar suas próprias informações, as quais podem ser trocadas com o sistema em desenvolvimento. A informação gerenciada por eles pode mudar independentemente do sistema que está sendo implementado.
- Sistemas atores estão *fora do escopo de desenvolvimento*, ou seja, a equipe não terá necessariamente acesso ao design interno desses sistemas, ou a habilidade de mudá-los. A equipe deve modelar a comunicação com o sistema ator usando as definições deste, porque elas não podem ser modificadas.

Algumas abordagens consideram que existem dois tipos de atores: *primários* e *secundários*. Atores primários seriam aqueles cujos objetivos o caso de uso está tentando satisfazer, enquanto os secundários são atores que apenas fornecem algum serviço para o processo em discussão. Exemplos de atores secundários são: procuradores, *web services* ou pessoas que devem fornecer informação ou confirmação, como, por exemplo, um funcionário que confirma que o cliente pagou em dinheiro. Neste livro, tal distinção não é feita: apenas atores *humanos* e atores *sistemas* serão identificados de forma diferenciada.

3.3 Casos de uso de sistema

Casos de uso de sistema diferem dos casos de uso de negócio (Seção 2.3) em alguns aspectos. Por exemplo, atores de negócio podem gastar dias ou mesmo semanas realizando um caso de uso de negócio, enquanto um caso de uso de sistema frequentemente é realizado em um período bem mais curto, usualmente em minutos, com um ou poucos atores interagindo com um sistema e obtendo um resultado completo e consistente para pelo menos uma de suas metas. Adicionalmente, casos de uso de sistema devem ser realizados sem interrupções enquanto casos de uso de negócio não têm restrições a este respeito.

Outra diferença fundamental entre um caso de uso de negócio e um caso de uso de sistema é que o primeiro é usualmente realizado por vários atores humanos, enquanto o segundo é normalmente realizado por poucos atores humanos (muitas vezes apenas um). O fato é que, se um caso de uso de sistema vai ser realizado por mais de um ator humano, então eles devem estar interagindo ao mesmo tempo com o sistema, e esta não é a situação mais comum. Usualmente, cada ator humano acessa o sistema no horário de sua melhor conveniência, verificando os dados e realizando as ações necessárias. Isso leva a uma sequência de casos de uso de sistema, e não a um único caso de uso, porque, como explicado anteriormente, um caso de uso de sistema não pode ser interrompido. Entretanto, algumas vezes, mais do que um único ator humano está disponível *on-line*: por exemplo, em um supermercado, quando um cliente está comprando produtos e um supervisor é chamado para realizar ações que o operador de caixa não tem permissão para fazer (como, por exemplo, cancelar uma venda). O ponto é que o supervisor deve estar disponível *on-line* e os outros atores devem aguardar até que ele apareça para realizar o procedimento.

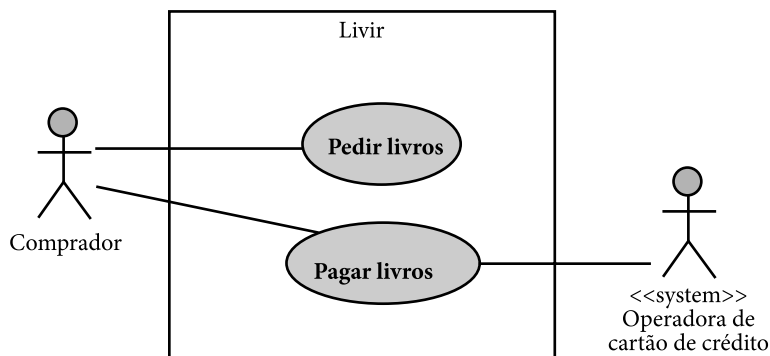
**Figura 3.1**

Diagrama de casos de uso de sistema.

No entanto, sistemas computacionais externos podem ser considerados atores *on-line* porque eles estão disponíveis continuamente. Por exemplo, pode-se assumir que uma operadora de cartão de crédito esteja *on-line* continuamente. No momento em que um comprador decidir fazer o pagamento com seu cartão de crédito, o ator de sistema *Operadora de cartão de crédito* vai estar disponível.

Um caso de uso de sistema de alto nível é apresentado simplesmente por um nome colocado dentro de uma elipse. Ele é usualmente associado a um ou mais atores, como mostrado na Figura 3.1.

No diagrama da Figura 3.1, as elipses representam os casos de uso de sistema. Por *default*, um ator é um papel representado por um humano. Outros tipos de atores são representados com o uso de estereótipos, como mostrado na Figura 3.1, em que *<<system>>* indica que a *Operadora de cartão de crédito* é um sistema externo e não um ser humano.

O diagrama de casos de uso é um diagrama UML bastante popular, mas também é frequentemente mal compreendido. É usual ver esses diagramas com dezenas de casos de uso e também com alguns de seus fragmentos acoplados. Entretanto, durante a concepção, é mais importante saber quais são os principais processos do sistema e não o seu detalhamento. Assim, não se recomenda a presença desses fragmentos no diagrama, nem o uso das associações *include* e *extend* entre casos de uso (as quais, algumas vezes, revelam parte da estrutura interna dos casos de uso). Questões relacionadas a fragmentos de casos de uso ou casos de uso incluindo ou estendendo outros podem ser deixadas para o momento em que estes processos de alto nível serão detalhados com o uso de diagramas de sequência de sistema (Seção 5.8).

Usualmente, ainda não há informação suficiente para descobrir todos os fragmentos neste momento. Por que deveriam apenas alguns fragmentos dos casos de uso ser mostrados e outros não? Melhor não mostrar nenhum durante a fase de Concepção. Isso evita que o diagrama acabe ficando com um número grande de elipses, o que prejudicaria sua compreensão. Assim, deve haver um critério preciso para decidir quais casos de uso de-

vem ser mantidos no diagrama para evitar, de um lado, um número grande de processos excessivamente detalhados, e, por outro lado, um número muito pequeno de processos, o que poderia fazer com que características importantes do sistema não aparecessem.

A regra consiste em considerar caso de uso apenas aqueles processos que podem ser executados isoladamente. Processos parciais que devem *necessariamente* ser executados durante outros processos não devem ser representados no diagrama de casos de uso de sistema.

Entretanto, mesmo seguindo essa norma, o número de casos de uso em um sistema do mundo real ainda pode ser muito alto, de forma que lidar com eles acaba ficando muito difícil. Para reduzir o número de casos de uso sem perder informação e precisão, uma segunda regra pode ser usada: agrupar casos de uso que são, de alguma forma, relacionados, especialmente se isso puder ser feito mais do que uma vez, seguindo um padrão. Por exemplo, poderia haver casos de uso como *Criar livro*, *Consultar livro*, *Atualizar livro* e *Deletar livro*, ou, simplesmente, um caso de uso chamado *Gerenciar livro*, que incluiria os quatro casos de uso mencionados antes. Isto é um padrão porque pode ser repetido para outros conceitos: *Gerenciar editora* poderia ser usado em vez de *Criar editora*, *Consultar editora*, *Atualizar editora* e *Deletar editora*. Este padrão é conhecido como *CRUD*, que é um acrônimo para os nomes das quatro operações em inglês: *Create*, *Retrieve*, *Update* e *Delete*.

As próximas subseções detalham essas regras e apresentam outros critérios para ajudar a escolher a melhor granularidade para casos de uso.

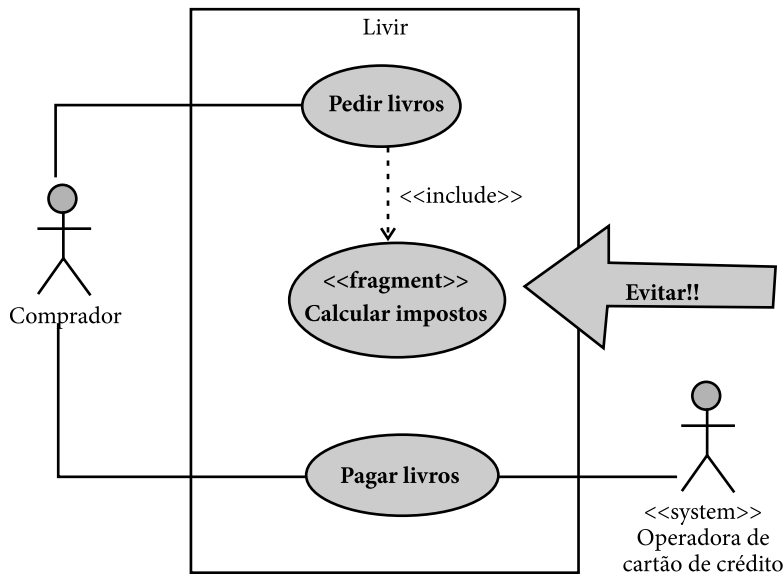
3.3.1 Sessão indivisível

Um bom caso de uso deve ser realizado em uma sessão indivisível de uso do sistema². Isso significa que ele deveria iniciar e terminar sem sofrer interrupção. Por exemplo, o registro de um pedido poderia ocorrer durante uma única sessão de uso do sistema, envolvendo a identificação de um cliente, a seleção de livros, a visualização de preços, o pagamento, a seleção de endereço para entrega etc. Cada um desses aspectos é um requisito funcional do sistema.

Casos de uso devem ser realizados como processos indivisíveis. Processos que só podem ocorrer no contexto de outros processos são apenas fragmentos, não casos de uso.

No caso do sistema *Livir*, calcular os impostos é algo que acontece apenas durante o processo de compra de livros. Pode-se supor que os requisitos do sistema não determinam que o cálculo de impostos deva ser considerado um processo independente (embora isso até pudesse acontecer, se os requisitos fossem outros). Nesse caso, *Calcular impostos* não é um caso de uso, e, em razão deste fato, não deveria ser incluído no diagrama. A Figura 3.2 ilustra esta situação que deve ser evitada – usar fragmentos do diagrama.

² Esta ideia segue a definição de um EBP (*Elementary Business Process* – Processo Elementar de Negócio) de Larman (2004). A definição de um EBP vem da área de engenharia de processo de negócio: “uma tarefa realizada por uma pessoa em um lugar em um tempo em resposta a um evento de negócio, a qual adiciona um valor mensurável ao negócio e deixa os dados em um estado consistente”.

**Figura 3.2**

Exemplo de um fragmento de caso de uso.

No entanto, *Pedir livros* pode ser considerado um caso de uso porque é um processo com início e final bem definidos, ocorrendo em um intervalo de tempo contíguo (sem interrupções) e que produz um resultado final consistente (o pedido é registrado).

No exemplo da Figura 3.1, existem dois cenários possíveis que poderiam ser investigados:

- O sistema só vai confirmar o pedido se o pagamento for feito *on-line* (ou seja, não é possível salvar o carrinho de compras). Nesse caso, os processos de pedir livros e pagar por eles seriam apenas fragmentos de um caso de uso que poderia ser chamado *Comprar livros*. Se este for o requisito real, então o diagrama deve ser alterado e os dois casos de uso substituídos por um único caso de uso chamado *Comprar livros*.
- O sistema vai registrar o pedido, mas não necessariamente o comprador precisa pagar no mesmo momento; ele pode guardar no carrinho de compras para finalizar o pedido em outro momento. Este é o cenário da Figura 3.1. Nesse caso, *Pedir livros* é um caso de uso e *Pagar livros* é outro caso de uso. Eles podem ser realizados imediatamente após o outro ou com um intervalo de até alguns dias; e, por causa disso, eles devem ser considerados casos de uso diferentes.

No entanto, livros não são automaticamente enviados assim que o pedido é pago: alguém deve ser responsável por verificar, de tempos em tempos, quais pedidos foram finalizados; esta pessoa vai coletar os livros no estoque e enviar o pacote. O processo de

envio pode acontecer assim que o pedido for finalizado, ou pode também acontecer no dia seguinte (ou mesmo seis a oito semanas depois). Este fato caracteriza *Enviar livros* como um caso de uso diferente. O caso de uso *Entregar livros* tem a compra dos livros como condição, mas a compra não é parte do processo de entrega: esta apenas acontece depois da compra. Assim, comprar e entregar, embora sejam sequenciais e partes do mesmo caso de uso de negócio, devem ser considerados casos de uso de sistema distintos, como mostrado na Figura 3.3, em que o caso de uso *Enviar livros* é introduzido.

3.3.2 Interativo

Um caso de uso deve ser interativo, o que significa que deve existir um ator para interagir com o sistema. Processos internos do sistema não são casos de uso, não importa quão complexos sejam. No entanto, uma simples consulta sobre a informação pode ser um caso de uso se ela for feita por um ator.

Os processos internos mencionados anteriormente podem ser parte de um caso de uso (como, por exemplo, *Calcular impostos*, que é parte de *Comprar livros*), ou requisitos suplementares (como, por exemplo, *Armazenar dados em um banco de dados relacional*). No caso de requisitos suplementares, eles são usualmente implementados por mecanismos internos que não dependem de interação com o usuário. Durante a fase de concepção, eles são registrados como especificações suplementares (Seção 3.4.8) ou requisitos não funcionais (anotações em casos de uso – Seção 3.4.5), de forma que o risco e o esforço implicados por eles não sejam subestimados.

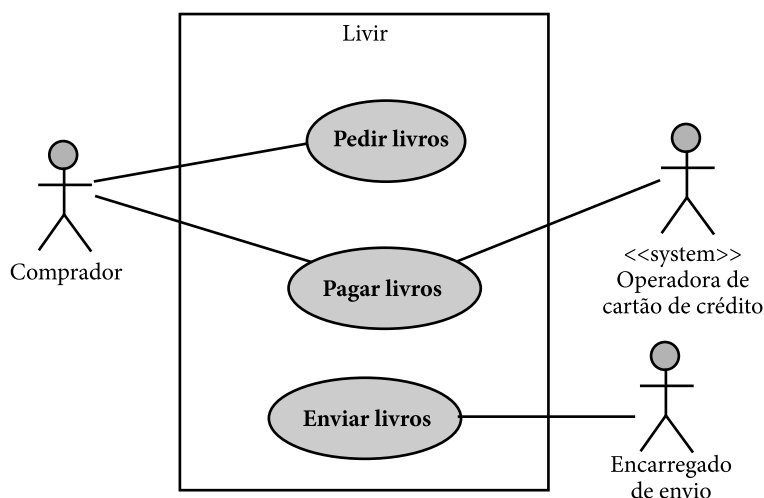


Figura 3.3

Casos de uso relacionados ao processo de negócio de venda de livros.

3.3.3 Resultado consistente

Um caso de uso deve produzir um resultado consistente, seja uma entrada completa ou transformação em uma peça de informação, ou simplesmente uma consulta na qual informação relevante é passada para o usuário. Um caso de uso não pode deixar a informação inconsistente ao seu final. Por exemplo, o registro de um pedido não pode ser concluído sem que o comprador e os livros que ele deseja tenham sido identificados, caso contrário a informação sobre o pedido estará incompleta em relação às regras do negócio; a livraria não pode reiniciar o pedido ou cobrar por ele sem saber quem é o comprador e quais são os livros solicitados.

Para decidir se um caso de uso tem um resultado consistente, deve-se pensar assim: apenas um processo completo pode ser um caso de uso de sistema, no sentido de que um usuário poderia ir até o computador, ligá-lo, realizar o processo e, ao final, desligá-lo, porque o processo estaria completo e algum objetivo de negócio teria sido obtido (alguma informação relevante foi recebida ou atualizada pelo usuário).

Isso exclui da definição de caso de uso os fragmentos como *Calcular impostos*, no caso do exemplo da livraria, isso porque os impostos são calculados dentro do processo de compra, e não como um processo isolado. Isso também exclui operações como *Login*, porque fazer login e em seguida desligar o computador não pode ser visto como um processo completo. Ele poderia apenas ser parte de um ou mais casos de uso (Figura 3.4).

Para evitar representar o *Login* como um caso de uso ou mesmo como parte de um caso de uso, pode-se simplesmente assumir que toda operação de sistema só pode ser executada por um usuário devidamente identificado e autorizado. Este mecanismo é parte do design tecnológico do sistema e não precisa ser considerado no modelo de casos de uso de alto nível.

No entanto, é possível que casos de uso completos ocorram dentro de outros casos de uso. Por exemplo, o processo de registrar um comprador pode ser considerado um caso de uso completo. Mas este processo também pode ocorrer durante a realização de um pedido de livros, especialmente se esta for a primeira vez que o comprador estiver usando o sistema, ou quando ele precisa atualizar seus dados (Figura 3.5).

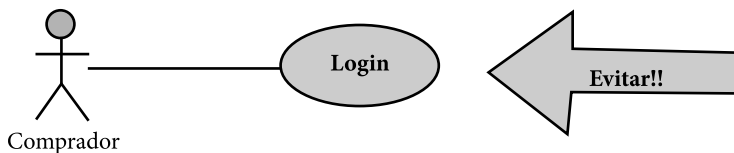
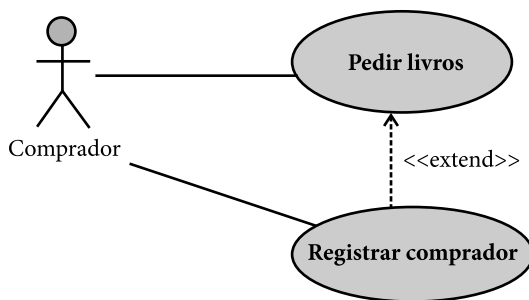


Figura 3.4

O *login* não deve ser considerado um caso de uso completo.


Figura 3.5

Casos de uso independentes podem estar relacionados.

A associação de dependência entre casos de uso é estereotipada com *<<extend>>* para indicar que o processo de pedir livros pode ocasionalmente ser estendido pelo processo de registrar um comprador. Ambos os processos podem ocorrer independentemente um do outro, mas o registro do comprador também pode ocorrer durante o pedido de livros.

Para evitar o abuso de *<<extend>>* no diagrama de casos de uso, é importante manter em mente que apenas casos de uso completos devem ser mantidos no diagrama. Fragmentos como aqueles mostrados nas Figuras 3.2 e 3.4 devem ser evitados, mesmo se forem pontos de extensão de um caso de uso. Tais fragmentos serão tratados adequadamente mais tarde, quando os casos de uso de alto nível forem detalhados (Capítulo 5). Na prática, neste ponto, qualquer associação entre casos de uso tais como *extend* ou *include* deve ser evitada no diagrama porque usualmente não adiciona informação útil para a fase de Concepção.

3.3.4 Essencial

Dois estilos para escrita de casos de uso podem ser identificados:

- *Casos de uso essenciais*, que não mencionam tecnologia de interface.
- *Casos de uso concretos* (ou *reais*), que são especificamente escritos para uma dada tecnologia de interface.

Durante a eliciação e a análise de requisitos, casos de uso de sistema são considerados requisitos e não design. É um erro comum incluir entre estes casos de uso ações que são puramente relacionadas com alguma tecnologia de interface (como *Abrir janela principal*, *Imprimir relatório* ou *Login*). Pessoas mais preparadas para lidar com os aspectos do design poderão decidir sobre essas ações mais adiante, depois que os requisitos forem descobertos.

Ambler (2000) aponta que modelos essenciais são mais flexíveis, deixando mais opções em aberto e que são mais capazes de acomodar mudanças na tecnologia. Ele também afirma que os modelos essenciais são mais robustos do que as representações concretas porque mais provavelmente permanecerão válidos caso haja alguma mudança na tecnologia de implementação.

Assim, casos de uso essenciais devem ser considerados a opção correta durante a fase de eliciação de requisitos, embora detalhes ou opções tecnológicas possam ser anotados para permitir análise de esforço e gerenciamento de risco. Mais adiante, neste livro (Seção 5.4.1), a diferença entre casos de uso essenciais e concretos será explicada em mais detalhe.

3.3.5 Alto nível

Durante a concepção, casos de uso são usualmente resumidos ou de alto nível, o que significa que sua descrição se dá apenas pelo seu nome ou, em alguns casos, por uma ou duas frases. Entretanto, esta não é a única forma pela qual um caso de uso pode ser descrito. Mais tarde, eles serão expandidos para conter mais detalhes sobre os requisitos (Capítulo 5). Cockburn (2001) identifica três tipos de casos de uso no que diz respeito ao nível de detalhe:

- *Resumido*: uma sinopse de um parágrafo sobre o caso de uso.
- *Casual*: escrito em um parágrafo com estilo de prosa simples. Ele provavelmente deixa de fora informações do projeto associadas ao caso de uso e também é menos rigoroso na sua descrição do que um caso de uso detalhado.
- *Detalhado*: expandido para incluir um fluxo principal e fluxos alternativos, bem como outras seções como pós-condições, pré-condições, interessados e variações tecnológicas.

Neste livro, espera-se que os casos de uso considerados durante a fase de Concepção sejam resumidos. Isso significa que usualmente apenas seu nome é suficiente para explicar seu significado. Entretanto, explicações adicionais são permitidas por conta da necessidade de compreensão dos requisitos. Adicionalmente, se alguns casos de uso-chave tiverem de ser expandidos para identificar riscos relacionados à sua complexidade inerente, isso também é aceitável nesta fase. Usualmente, casos de uso detalhados somente serão úteis após a Concepção, quando os requisitos precisam ser detalhados adequadamente.

3.3.6 Fronteira do sistema

Uma das decisões que a equipe deve tomar quando estiver trabalhando com os casos de uso de sistema é onde colocar a *fronteira de sistema*. Graficamente, a fronteira é apenas um retângulo que é colocado no diagrama. Dentro dele estão os casos de uso e fora, os atores. Nos diagramas de casos de uso de negócio, a fronteira de sistema representa os limites da organização (empresa, departamento etc.). Aqui ela representa os limites de um sistema computacional.

Foi tomada uma decisão anteriormente sobre quais trabalhadores de negócio incluir nos limites de automação. Um exemplo foi o funcionário que ajudava o cliente a comprar livros. Naquele caso, quando foi feita a modelagem dos casos de uso de sistema, o *Funcionário* desapareceu, e o caso de uso passou a ser executado pelo próprio cliente.

Entretanto, caso esta não fosse uma livraria virtual, mas uma tradicional, na qual os clientes fazem compras na loja, o funcionário deveria ser mantido no diagrama de casos de uso? Para que um caso de uso seja o mais essencial possível, recomenda-se que apenas os atores que realmente têm interesse nele sejam mantidos no diagrama. O funcionário do exemplo é apenas um procurador para o cliente; ele não tem objetivos pessoais no processo de compra de livros, atuando simplesmente no lugar da interface do sistema. Seja a livraria virtual ou não, o caso de uso essencial deveria ser o mesmo, porque a mesma informação é trocada entre o comprador e o sistema; o funcionário apenas transmitiria a informação que recebesse. Nesse caso, o único ator deveria ser o comprador e o funcionário não deveria sequer aparecer no diagrama. Dessa forma, a análise produziria casos de uso que seriam independentes de tecnologia.

Alguém poderia perguntar o seguinte: e se o funcionário precisasse indicar que ele ajudou na venda, de forma que uma percentagem fosse paga a ele? Nesse caso, o que se tem é um caso de uso que é diferente daquele mencionado no último parágrafo. Aqui, tanto o comprador quanto o funcionário *devem* ser atores: o comprador está interessado nos livros e o funcionário, na comissão. Além disso, se o funcionário puder executar ações que não são permitidas ao comprador (por exemplo, alterar o preço de um produto), então ele deveria ser considerado um ator e não deveria ser removido do diagrama.

3.4 Como encontrar casos de uso de sistema no modelo de negócio

Para descobrir casos de uso e atores de sistema, a equipe pode examinar o diagrama de casos de uso de negócio com a fronteira de automação definida (escopo de automação).

Primeiramente, os atores que estão realmente interessados nos processos a serem automatizados devem ser identificados. Eles são os atores de negócio que interagem com casos de uso de negócio que estejam dentro do escopo de automação, e os trabalhadores de negócio que interagem com tais casos de uso, mas que não estejam eles próprios dentro do escopo de automatização. No caso da Figura 2.11, os seguintes atores de sistema podem ser identificados:

- *Comprador, Editora e Operadora de cartão de crédito*, porque são atores de negócio que interagem com casos de uso que serão automatizados.
- *Gerente de compras*, porque é um trabalhador de negócio que não será automatizado, mas que interage com um caso de uso que vai ser.

No entanto, os seguintes não são considerados atores de sistema:

- *Funcionário*, porque é um trabalhador de negócio cujo papel será automatizado.
- *Departamento de marketing*, porque é um trabalhador de negócio que não interage com caso algum de uso dentro do escopo de automatização.

Trabalhadores de negócio cujas funções serão parcial ou totalmente automatizadas pelo sistema serão importantes fontes de informação, porque saber como eles

realizam seus deveres atualmente pode ser crucial para entender como o sistema deve funcionar. Mais do que isso, saber como eles realmente executam seu trabalho pode ser valioso porque algumas vezes a prática é diferente do que está escrito nos livros de procedimento.

Outra fonte de requisitos são os atores de negócio que se tornarão atores de sistema. Entretanto, eles nem sempre estarão disponíveis para eliciação de requisitos e, nesse caso, eles devem ser substituídos por especialistas de sistema. Por exemplo, pode não ser possível entrevistar o cliente de uma loja que ainda não exista. Assim, alguém que entenda do negócio e possa explicar como se espera que ele funcione será uma pessoa-chave para obter os requisitos corretos. Se as editoras e operadoras de cartão de crédito também não estiverem disponíveis, ainda seria possível examinar a documentação existente sobre os protocolos de comunicação e os serviços oferecidos de forma que os requisitos de interfaceamento com eles possam ser descobertos. No caso do cliente, entretanto, esta possibilidade não é viável.

Uma vez que os atores de sistema tenham sido identificados a partir dos casos de uso de negócio, será possível observar os diagramas de atividade e de máquina de estados que foram produzidos durante a análise de negócio para verificar quais atividades realizadas por estes podem ser consideradas casos de uso de sistema.

Pela análise das atividades da Figura 2.10, pode-se ver que as atividades realizadas pelo *Comprador* são:

- *Pedir livros.*
- *Pagar livros.*

A única atividade da *Editora* é *Enviar livros* e a única atividade da *Operadora de cartão de crédito* é *Confirmar pagamento*.

Nem toda atividade do *Funcionário* original se torna um caso de uso. Como visto na Figura 2.7, o funcionário foi dividido em dois atores: *Gerente de compras*, que não será automatizado, e *Funcionário*, que será automatizado. As atividades realizadas pelo papel de *Gerente de compras* podem tornar-se casos de uso de sistema. Entretanto, as atividades realizadas pelo trabalhador de negócio *Funcionário*, que está sendo automatizado, tornam-se ações internas do sistema. Por exemplo, as atividades *Registrar pedido* e *Informar disponibilidade e total* na Figura 2.10 não se tornam casos de uso de sistema; elas são apenas parte do caso de uso *Pedir livros*, iniciado pelo comprador. Entretanto, *Pedir livros ao fornecedor* é um caso de uso de sistema cujo ator é *Gerente de compras*.

Em relação aos processos que serão identificados como casos de uso, consideram-se duas questões:

- As atividades conectadas por um fluxo devem ocorrer de forma imediata ou podem ocorrer em momentos diferentes? No exemplo, as atividades *Pedir livros* e *Pagar livros* devem ocorrer em uma única sessão de uso do sistema ou elas podem ser realizadas

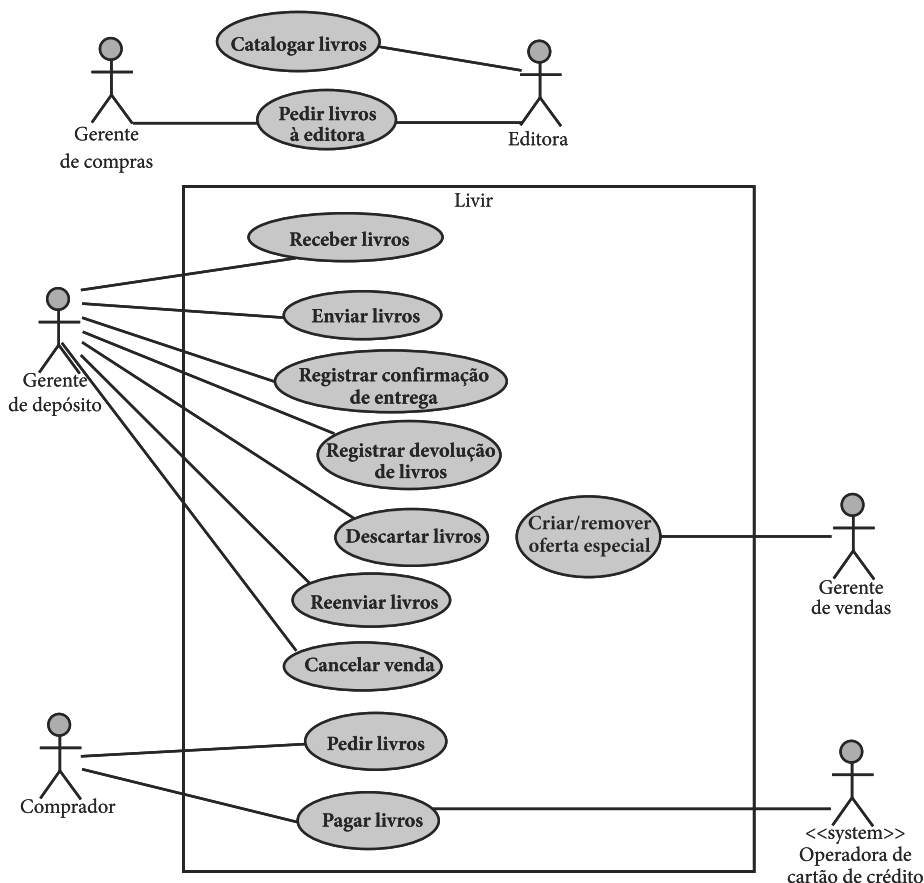
em momentos diferentes? A resposta dependerá da forma como o negócio é organizado, isto é, dependerá das regras de negócio. Se a empresa decidir que o pedido só é recebido depois que o pagamento é feito, então essas atividades formam um único caso de uso. Entretanto, se a empresa decidir que é possível registrar um pedido (salvar carrinho de compras) e pagar por ele em outro dia, então existem dois casos de uso distintos.

- As atividades selecionadas são completas e consistentes? Por exemplo, *Enviar livros* é uma boa descrição para a atividade realizada pela editora? Por similaridade com o processo realizado pelo cliente, pode-se inferir que a editora vai receber um pedido e que então vai enviar os livros em estoque. Entretanto, o pagamento na relação entre empresas normalmente não é feito imediatamente – uma fatura é gerada e paga pela livraria em um prazo predefinido (por exemplo, 30 dias após a compra).

Neste ponto, o diagrama da Figura 2.10 deveria ser revisado. A editora não vai simplesmente *Enviar livros*. Ela vai receber um pedido, enviar uma fatura para pagamento e, finalmente, enviar os livros que estiverem disponíveis. Os livros devem ser recebidos e registrados por algum trabalhador na livraria. Este trabalhador será um novo ator que poderia ser chamado de *Encarregado do depósito*.

Uma questão recorrente é se atores correspondem a perfis de segurança. Este não é necessariamente o caso. O objetivo por trás da identificação e modelagem dos atores está mais relacionado ao processo de encontrar e organizar requisitos do que ao processo de disponibilizar acesso ao sistema. Um bom design de software vai tratar a permissão de acesso de forma dinâmica, permitindo que perfis de usuário sejam criados, e permissões sejam associadas aos diferentes perfis de usuários individuais. Como este aspecto do sistema não é dependente de domínio, não é adequado que ele seja modelado junto ao domínio. Em outras palavras, atores de casos de uso não devem ser considerados perfis de segurança. Estes perfis serão criados dinamicamente com o uso de um mecanismo genérico independentemente de domínio.

O diagrama de máquina de estados apresentado na Figura 2.15 é também particularmente útil em termos dos casos de uso de sistema que precisam ser analisados. Cada mudança de estado em um livro (principal objeto de negócio) deve ser feita por alguém, e este “alguém” é um ator que executa o caso de uso. Assim, a partir das transições daquele diagrama, um novo conjunto de casos de uso pode ser descoberto, conforme mostrado na Figura 3.6.

**Figura 3.6**

Modelo de casos de uso de sistema completado com informação descoberta no diagrama de máquina de estados da Figura 2.15.

Na Figura 3.6, os casos de uso *Catalogar livros* e *Pedir livros à editora* são mantidos fora do escopo do sistema *Livir*. Isso se deve ao fato de que foi descoberto neste ponto que tais casos de uso seriam realizados com o sistema da editora, e não com o sistema sendo desenvolvido. Como eles não estão incluídos no sistema da livraria, podem ser mantidos no diagrama, mas fora da fronteira do sistema, porque estão fora do escopo e não serão implementados.

Após estar claramente entendido que não serão implementados, eles podem ser removidos totalmente do diagrama para simplificá-lo e para evitar confusão com interessados que possam não entender que o que está fora da fronteira do sistema não será implementado. No caso, os nomes desses casos de uso podem ser referenciados em uma lista de *casos de uso fora do escopo*, para fins de registro.

Os casos de uso associados com o novo ator *Gerente de depósito* são necessários para cobrir a maior parte das transições indicadas no diagrama de máquina de estados da Figura 2.15. É interessante notar que algumas transições são realizadas fora do escopo da

empresa. Por exemplo, quando um comprador decide devolver um livro, isso não é feito por meio do sistema da empresa; ele simplesmente envia o livro de volta, usualmente por correio. A informação referente à devolução apenas chega à livraria quando ela recebe o livro. Assim, o caso de uso referenciado aqui é *Registrar devolução de livro*, e não *Devolver livro*, porque o registro é feito pelo gerente de depósito, e não pelo comprador. Isso é similar ao registro da chegada de livros, representada pelo caso de uso *Receber livros*, porque quando a editora envia os livros isso não é diretamente registrado no sistema da livraria (a não ser que os dois sistemas sejam integrados). Apenas a chegada pode ser registrada pela livraria. Esta é uma diferença fundamental entre um caso de uso de negócio e um caso de uso de sistema: em um caso de uso de negócio, o processo como acontece no mundo real pode ser descrito como parte do caso de uso, mas quando o foco está no sistema, apenas processos que envolvam o sistema podem ser representados.

3.5 Requisitos

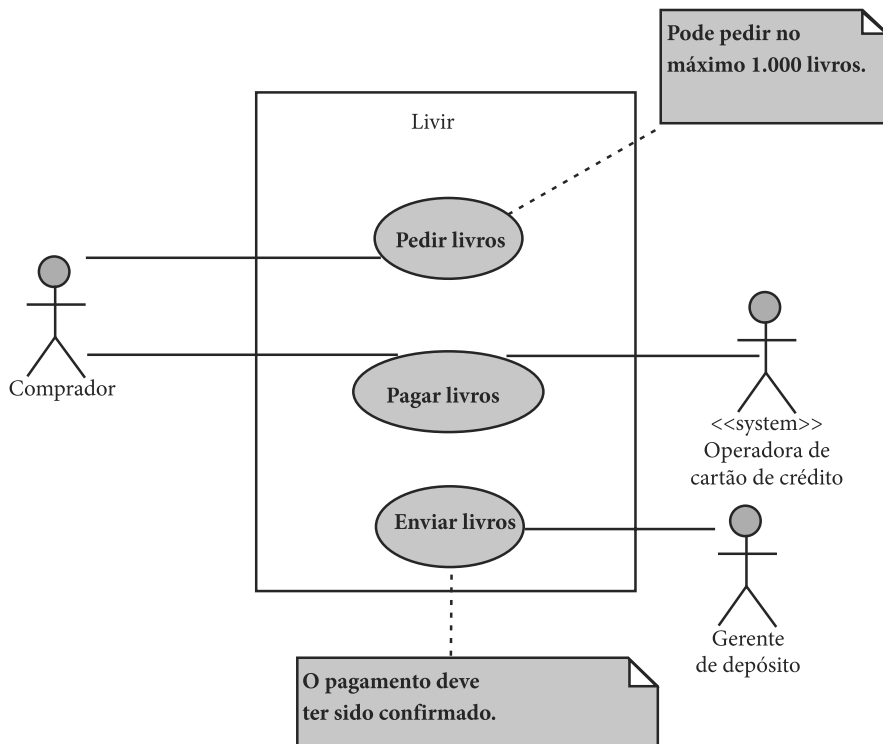
Identificar casos de uso de sistema é uma atividade parte da disciplina de *Requisitos* do Processo Unificado. A eliciação e a análise de requisitos consistem em uma parte significativa da fase de Concepção. A equipe pode e deve usar todas as fontes disponíveis para identificar requisitos (especialistas, usuários, documentos, interfaces, literatura etc.) e, para cada fonte, um conjunto de funções que o sistema deve realizar pode ser identificado.

3.5.1 Eliciação de requisitos

Eliciação de requisitos corresponde à pesquisa por informação sobre as funções que o sistema deve realizar, e pelas restrições sob as quais o sistema deve operar. Na abordagem apresentada neste livro, elas serão registradas na forma de casos de uso de sistema.

Outra forma de registrar requisitos seria pelo uso de um documento que consiste em uma lista de requisitos funcionais, possivelmente acompanhada de uma lista de restrições (Sommerville, 2006). A vantagem de usar casos de uso em vez de uma lista de funções consiste no fato de que um bom caso de uso tem uma granularidade bem definida, o que permite a geração de uma lista de requisitos de alto nível que é muito mais compreensível do que uma lista de funções individuais. Usualmente, o número de casos de uso de alto nível é muito mais baixo do que o número de funções individuais. As funções individuais vão aparecer na expansão dos casos de uso, quando os casos de uso de alto nível terão sua estrutura detalhada.

No caso do exemplo *Livir*, a eliciação de requisitos permite que a equipe descubra que o sistema deve controlar a compra e venda de livros, receber pagamentos, permitir o registro de livros danificados, gerar relatórios de vendas, verificar se livros estão disponíveis no estoque etc. Essas operações e muitas outras constituem a funcionalidade do sistema, e este é o motivo pelo qual tais requisitos são chamados de funcionais. Essas funções serão incorporadas em um ou mais casos de uso.

**Figura 3.7**

Requisitos não funcionais anotados em casos de uso.

No entanto, durante a eliciação de requisitos, o analista pode encontrar regras de negócio ou restrições sobre como as funções devem ser realizadas pelo sistema. Por exemplo, uma regra de negócio poderia estabelecer que a livraria apenas enviasse livros após a confirmação do pagamento. Este tipo de regra é um requisito não funcional que pode ser registrado como uma anotação ou comentário no próprio caso de uso, para que possa ser lembrada e verificada quando o caso de uso for expandido (Figura 3.7). Alternativamente, essas regras poderiam ser registradas de forma separada, como uma lista numerada ou uma planilha com referência aos casos de uso a partir de um número identificador único.

Como pode ser visto na Figura 3.7, o diagrama de casos de uso com notas rapidamente se tornaria muito complexo para ser útil. É por isso que se recomenda que tais notas sejam registradas dentro da especificação do caso de uso ou em um documento à parte. A maioria das ferramentas *Computer Aided Software Engineering*³ (CASE) permite que um elemento de diagrama tal como um caso de uso tenha uma janela de especificação como a mostrada na Figura 3.8. Uma exceção a esta regra poderia ser o caso em que uma anotação seja absolutamente necessária para ajudar na compreensão do diagrama. Mas este raramente é o caso.

³ Engenharia de Software Apoiada por Computador.

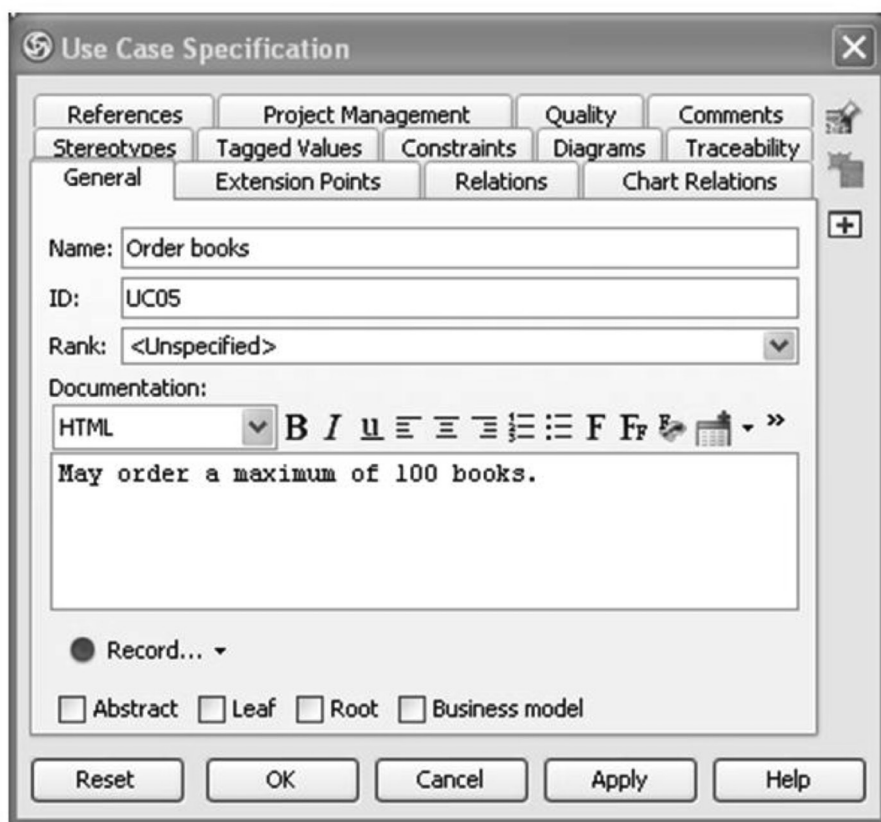


Figura 3.8

Exemplo de uma janela de especificação para um caso de uso. Imagem produzida com a ferramenta CASE Visual Paradigm™ Community Edition.

3.5.2 Eliciar requisitos não é design!

Um sistema sendo analisado é como uma floresta. Explorar uma floresta desconhecida não seria efetivo caso se iniciasse examinando cada planta e animal. Há um ditado que diz que algumas pessoas não conseguem ver a floresta por causa das árvores. A floresta é o sistema e as árvores são os requisitos. Apenas ao final do processo uma equipe poderá dizer que adquiriu conhecimento sobre as menores partes. Mas, antes disso, uma visão do todo deve ser tomada, e apenas depois disso é que os detalhes podem ser estudados.

Assim, a fase de Concepção deve fornecer uma visão do todo – de forma que o mais importante possa ser visto primeiro – e então o todo pode ser dividido em partes de forma que os detalhes possam ser analisados e, finalmente, uma solução projetada. A organização das iterações nas fases de Elaboração e Construção corresponde à divisão da floresta em setores de forma que se possa olhar um setor de cada vez e dessa forma ser capaz de lidar com a complexidade inerente. Assim, um dos objetivos finais da fase de concepção consiste

em organizar a divisão do trabalho em termos de casos de uso, os quais são explorados durante as iterações seguintes.

Durante a concepção, a eliciação de requisitos deve ser rápida e genérica. O jeito certo de fazer isso é olhar para a extensão dos requisitos, e não para seus detalhes. O analista deve entender a extensão do que o sistema deve fazer, sem detalhar como ele vai fazer isso. Apenas durante as iterações da Elaboração essa análise de requisitos será aprofundada.

O tempo usado para eliciação de requisitos deve ser gasto com descoberta, não com invenção. Durante este período, a equipe de analistas, com os clientes, usuários e outros interessados, deve tentar listar a maioria das capacidades e restrições sem preocupação em detalhá-las. Detalhes sobre requisitos serão convenientemente acomodados nas próximas iterações.

Deve ficar claro também que requisitos são algo que o cliente pediu, e não algo que a equipe projeta. Alguns analistas podem confundir coleta de requisitos – isto é, a memória das demandas do cliente – com um design de sistema preliminar. Um exemplo desse tipo de confusão é um conjunto de requisitos que envolvem o design de bancos de dados relacionais. A não ser que existam sistemas legados que devem estar compatíveis com o novo sistema, como se pode justificar que um conjunto de tabelas relacionais seja uma demanda do cliente? Isso poderia eventualmente ser possível com algum cliente sofisticado, com formação na área de computação, mas esta não é a regra geral. Tabelas de bancos de dados são parte do domínio da *solução*, e não do domínio do problema. O analista deve procurar os requisitos que correspondem às necessidades e aos objetivos do cliente em termos de informação. Mais tarde, ele pode decidir se essa informação vai ser armazenada em um banco de dados relacional ou em alguma outra estrutura.

3.5.3 Desafios dos requisitos

O documento de requisitos, que pode ser formado pelo diagrama de casos de uso com anotações e pelas especificações suplementares, deve registrar as capacidades do sistema e as condições sob as quais ele trabalha. Os desafios referentes aos requisitos são, pelo menos, os seguintes (Pressman, 2010):

- Como *descobrir* requisitos.
- Como *comunicar* requisitos às outras fases e equipes do projeto.
- Como se *lembrar dos* requisitos durante o desenvolvimento para verificar se eles foram todos implementados.
- Como *gerenciar* a mudança dos requisitos.

Seria inútil desenvolver um belo diagrama de casos de uso e mais tarde não ser capaz de saber se os requisitos incorporados foram incluídos no design ou não. A existência de mecanismos automáticos para conduzir esta verificação é crucial. Portanto, é importante manter as relações de rastreabilidade entre casos de uso e outras partes do design. Os capítulos seguintes mostram como essas relações entre artefatos de design podem ser obtidas.

É necessário manter em mente que os requisitos necessariamente mudam durante o desenvolvimento de um sistema. Assim, a mudança deve ser gerenciada, e não evitada, após a Concepção.

Algumas vezes, os requisitos mudam depois que o sistema é implantado. Condições de contexto, regulamentos, políticas de empresa ou métodos de trabalho podem mudar a qualquer hora. Embora a análise não possa prever essas mudanças, mecanismos para acomodá-las podem ser criados de forma a facilitar o processo de mudança quando ele for necessário. Existem padrões de design específicos para lidar com essas instabilidades de requisitos (como, por exemplo, o padrão *Estratégia*, que será apresentado na Seção 7.6). As mudanças são totalmente imprevisíveis. Se o sistema não for estruturado para acomodar as mudanças nos requisitos, elas provavelmente serão muito difíceis de implementar.

Este tipo de situação faz com que processos de análise e design *orientados a requisitos* (Alford, 1991) sejam inadequados para a maioria dos sistemas. O uso de requisitos como base para suportar a arquitetura do sistema é como construir uma casa em areia movediça; quando os requisitos mudam, a estrutura sofre. Entretanto, o Processo Unificado (UP) usa uma base muito mais estável para a arquitetura, a qual é formada por classes e componentes, que encapsulam informação e comportamento.

Essas classes implementam funcionalidades que, combinadas, permitem a implementação dos requisitos. Se os requisitos mudam, as combinações mudam, mas não a estrutura básica. Este tipo de arquitetura segue o *princípio aberto-fechado* (Meyer, 1988), no sentido de que ela está sempre fechada para modificação (ela funciona), mas aberta para extensão (ela pode acomodar novas funcionalidades).

É importante rastrear a origem de cada requisito (por exemplo, um ator de negócio, trabalhador de negócio, o próprio cliente ou mesmo um especialista de domínio), porque é necessário validar os requisitos com estas fontes, verificando se estão bem escritos, completos e consistentes.

Algumas vezes também pode ocorrer que diferentes pessoas ou departamentos apresentem distintas especificações para os mesmos requisitos. Nesse caso, é necessário produzir um acordo entre eles, ou identificar quem tem mais autoridade para determinar a forma aceitável do requisito.

3.5.4 Requisitos funcionais evidentes e ocultos

Requisitos funcionais podem ser opcionalmente identificados como *evidentes* ou *ocultos* (Gause; Weinberg, 1989):

- Requisitos funcionais *evidentes* são funções realizadas com o conhecimento do usuário. Estes requisitos usualmente correspondem a trocas de informação entre o usuário e o sistema, como consultas e entradas de dados, os quais fluem por meio da interface do sistema.

- Requisitos funcionais *ocultos* são funções realizadas pelo sistema sem o conhecimento explícito do usuário. Usualmente, estas funções são operações matemáticas e atualizações de dados realizadas pelo sistema sem o conhecimento explícito do usuário, mas como consequência de outras funções realizadas por ele.

Requisitos ocultos são realizados internamente pelo sistema. Assim, embora eles não apareçam explicitamente como casos de uso, eles devem estar adequadamente associados a eles para serem lembrados no momento do design e implementação. Assim, eles podem também ser adicionados como anotações aos casos de uso.

Um exemplo de requisito evidente é a produção de um relatório de livros mais lidos, que é algo que um gerente poderia querer. Um exemplo de requisito oculto é aplicar uma política de desconto a uma venda. Nesse caso, o usuário não solicita explicitamente que o sistema realize a operação. Como esta é uma atividade que o sistema realiza automaticamente durante a realização de outra operação, ela é um requisito oculto.

3.5.5 Requisitos não funcionais

Requisitos não funcionais são restrições ou qualidades que podem estar ligadas a funções específicas de um sistema (por exemplo, “um pedido não pode conter mais do que 100 livros”, “a transação de pedido deve ser preservada mesmo se as comunicações caírem” etc.), e, assim, elas podem ser tratadas como anotações aos casos de uso que incorporam as respectivas funções. Algumas vezes, entretanto, requisitos não funcionais podem ser gerais, isto é, não necessariamente ligados a uma função (por exemplo, “o sistema deve ser implementado em Java”), e, nesse caso, eles devem aparecer no documento de especificações suplementares. Restrições e qualidades que são especificamente ligadas a uma função são chamadas de *requisitos não funcionais* e restrições e qualidades gerais são chamadas de *requisitos suplementares*.

Há dois tipos de requisitos não funcionais:

- *Restrições lógicas*: regras de negócio associadas a uma função. Por exemplo, durante o registro de uma venda, uma série de restrições poderia ser considerada, tais como não fechar a venda até que a operadora de cartão de crédito confirme o pagamento, ou não fechar uma venda se a última entrega para o mesmo endereço retornou por conta de ele ser inválido.
- *Restrições tecnológicas*: restrições e qualidades relacionadas à tecnologia usada para realizar a função, como, por exemplo, a interface com usuário, o tipo de protocolo de comunicação, restrições de segurança, tolerância a falhas, e assim por diante.

Por exemplo, estabelecer que a interface com usuário para realizar um pedido deve seguir um padrão de design baseado em um fluxo sequencial de telas é uma restrição tecnológica (restrição de interface) sobre a forma como a função vai ser realizada. Outro exemplo de requisito não funcional é: “o pagamento não deve demorar mais do que 5 segundos”. Esta é uma restrição tecnológica relacionada ao desempenho do sistema e deve

afetar a forma como o designer vai pensar sobre o mecanismo de comunicação com a operadora de cartão de crédito. Nesse caso, o design do sistema deve considerar seriamente o uso de uma conexão de banda larga dedicada para as operadoras.

Um requisito não estabelece como as restrições vão ser implementadas, ele apenas as exige. O design e a implementação do sistema devem atender aos requisitos de um jeito ou de outro, ou, caso contrário, o analista terá de negociar com o cliente por alguma flexibilidade nos requisitos.

3.5.6 Requisitos não funcionais permanentes e transitórios

Uma das características mais fundamentais dos requisitos não funcionais e suplementares é se eles são *permanentes* ou *transitórios*.

Requisitos não funcionais ou suplementares podem ser considerados permanentes (não se espera que eles mudem) ou transitórios (espera-se que eles mudem) dependendo de uma decisão tomada pelo cliente. Permanência ou transitoriedade não são características intrínsecas de um requisito: elas são decididas conforme a conveniência. Os mesmos requisitos dados podem ser considerados permanentes ou transitórios dependendo do que se deseja em termos do custo de desenvolvimento ou manutenção do software.

Se o investimento for feito em design flexível, de forma que a maioria das restrições seja transitória, menos esforço será gasto durante a manutenção para acomodar eventuais mudanças. Entretanto, o custo desse design flexível pode tornar-se proibitivo para alguns projetos. Sempre será uma boa ideia ponderar quais restrições deveriam realmente ser consideradas requisitos transitórios.

Por exemplo, um requisito suplementar poderia estabelecer que o sistema Livir deveria trabalhar com uma única moeda: o real. Se este requisito for considerado permanente, então o sistema será projetado com uma única moeda (“real” poderia mesmo ser um tipo de dado primitivo usado para definir variáveis e atributos⁴). Entretanto, se o requisito for considerado transitório, então mesmo se não houver outra moeda sendo usada hoje o sistema inteiro deve ser preparado para acomodar novas moedas a serem definidas no futuro, ou até mais de uma moeda de cada vez.

As consequências de decidir que um requisito é *permanente* são as seguintes:

- É mais barato e rápido desenvolver o sistema.
- É mais caro e difícil dar manutenção no sistema se, por acaso, os requisitos mudarem no futuro.

No entanto, decidir que um requisito é *transitório* tem as seguintes consequências:

- É mais caro e complexo desenvolver o sistema (ele deveria acomodar funcionalidades para mudar a moeda, por exemplo).
- É mais fácil e barato dar manutenção no sistema (se a moeda mudar, o sistema já estará pronto para acomodar isso com uma simples reconfiguração).

⁴ No caso de linguagens que já possuem o tipo “real” para definir números de ponto flutuante, um nome alternativo para designar a moeda talvez devesse ser escolhido.

Assim, não é a natureza do requisito não funcional que vai decidir se ele é permanente ou transitório. É o cliente, com a ajuda da equipe, que deve tomar esta decisão. A situação ideal seria listar os requisitos de maior importância (aqueles que realmente se espera que mudem em um futuro próximo com grande impacto para o sistema) e considerá-los transitórios, enquanto os outros são deixados como permanentes.

3.5.7 Requisitos obrigatórios e desejáveis

Requisitos também podem ser *obrigatórios* ou *desejáveis*⁵, isto é, aqueles que devem ser obtidos de qualquer maneira e aqueles que devem ser buscados a não ser que possam prejudicar o andamento do projeto.

No caso dos requisitos funcionais, esta classificação usualmente indica prioridade de desenvolvimento. Se há flexibilidade no contrato tal que apenas os requisitos mais importantes sejam implementados caso não haja tempo para implementar todos, então a equipe deve saber quais requisitos são obrigatórios.

Entretanto, se a equipe é capaz de fazer uma boa estimativa do esforço necessário para desenvolver o sistema, e se tem um bom histórico de estimações precisas, haverá menos motivação para tal distinção em relação aos requisitos funcionais, porque se espera que todos os requisitos possam ser implementados a tempo.

No entanto, requisitos não funcionais e suplementares são muito mais imprevisíveis do que os funcionais em relação à estimativa de esforço. Assim, em alguns casos, pode ser necessário considerar esses requisitos com alguma flexibilidade.

Nesse caso, algumas restrições são definidas de forma que elas tenham de ser obtidas por quaisquer meios e outras podem ser consideradas apenas desejáveis, com algum tempo de projeto alocado para que elas possam ser buscadas.

Por exemplo, no caso do sistema Livir, o requisito para usar uma interface Web poderia ser considerado um requisito suplementar obrigatório. Nesse caso, outras soluções não seriam aceitáveis. Entretanto, o acesso adicional a partir de telefones celulares poderia ser considerado um requisito desejável, porque este acesso não é absolutamente necessário para o efetivo sucesso do sistema.

Atualmente, com a formalização dos contratos de desenvolvimento de software, existe cada vez menos flexibilidade em relação a requisitos desejáveis. Em muitos casos, os desenvolvedores devem definir quais requisitos serão implementados, quanto tempo vai levar e quanto isso vai custar. Sair desta linha pode exigir o pagamento de multas ou mesmo o cancelamento do projeto.

3.5.8 Requisitos suplementares

Requisitos suplementares são todo tipo de restrições ou qualidades relacionadas ao sistema como um todo, e não apenas a funções individuais. Por exemplo, um requisito suplementar

⁵ Outra opção consiste em usar a escala MoSCoW: Must, Should, Could e Would.

pode estabelecer que o sistema deve ser compatível com um determinado banco de dados legado, ou ser implementado com uma determinada linguagem de programação, ou, ainda, seguir um determinado design de interface (*look and feel*).

Deve-se tomar cuidado com a definição dos requisitos suplementares. Um requisito como “o sistema deve ser fácil de usar” não é suficientemente claro. Seria melhor dizer que “um usuário novo deve ser capaz de completar as tarefas sem erro na primeira tentativa”. Isso dá uma ideia mais precisa sobre o que deve ser projetado para atender ao requisito.

Requisitos não funcionais e suplementares também podem ser identificados em diferentes grupos, como, por exemplo, interface, implementação, desempenho, tolerância a falhas etc. O objetivo de fazer tais distinções está em permitir uma organização mais adequada.

Embora a maioria dos praticantes do UP possa escolher o sistema de classificação FURPS+ (Grady, 1992) para organizar requisitos suplementares, uma fonte mais atualizada para a classificação de tais requisitos por grupos de qualidade é a norma ISO/IEC 25010⁶, como mostra a Tabela 3.1.

Tabela 3.1 Classificação de requisitos suplementares baseada na ISO/IEC 25010:2011 e questões geradoras de requisitos.		
Característica 25010	Subcaracterística 25010	Questões geradoras de requisitos
Adequação funcional (<i>functional suitability</i>)	Compleitude funcional	São estas todas as funções necessárias que devem ser implementadas? Algumas delas podem ficar de fora do escopo do sistema? Algumas funções podem ser implementadas em projetos futuros?
	Funcionalidade apropriada	Em que grau as atividades do usuário devem ser facilitadas pelo sistema?
	Corretude funcional (acurácia)	Existem especificações de acurácia, ou seja, há uma precisão desejável para os dados? Existem limites toleráveis para imprecisão?
Confiabilidade (<i>reliability</i>)	Maturidade	Algum cuidado especial deve ser tomado para evitar que o sistema apresente defeitos durante seu uso? Partes críticas do sistema devem ser definidas por especificação formal? Quais a intensidade e o tipo de testes que devem ser realizados para assegurar que o sistema esteja livre de defeitos?
	Disponibilidade	Qual o grau de disponibilidade requerido para o sistema? Quantos acessos simultâneos devem ser suportados? Quantas horas por dia? Quantos dias por ano?
	Tolerância a falhas	Como o sistema deve reagir no caso de anomalias externamente provocadas, como falha de comunicação?
	Recuperabilidade	O sistema deve ser recuperar automaticamente no caso de desastre? Sob quais circunstâncias dados perdidos e processos abortados devem ser recuperados?

⁶ Disponível em: <www.iso.org/iso/catalogue_detail?csnumber=35733>.

Tabela 3.1 Classificação de requisitos suplementares baseada na ISO/IEC 25010:2011 e questões geradoras de requisitos.

Característica 25010	Subcaracterística 25010	Questões geradoras de requisitos
Usabilidade (<i>usability</i>)	Apropriação reconhecível	De que maneira o software deve se apresentar a um potencial usuário de forma que ele possa reconhecer sua aplicabilidade? Como o software deve ser empacotado?
	Inteligibilidade	Como os conceitos inerentes ao software são apresentados ao usuário para que ele possa se tornar competente no seu uso?
	Operabilidade	Em que grau o produto deve ser fácil de usar e controlar? Que tipo de ajuda o sistema deve prover? Que formas de documentação e manuais devem estar disponíveis? Como eles vão ser produzidos? Que tipo de informação eles devem apresentar?
	Proteção contra erro de usuário	Que tipo de proteção contra erro do usuário é necessária?
	Estética de interface com usuário	Quais padrões de design de interface com usuário serão usados para fornecer um visual prazeroso e uma interação satisfatória?
	Acessibilidade	Em que grau o produto deve ser projetado para atender pessoas com necessidades especiais?
Eficiência de desempenho (<i>performance efficiency</i>)	Comportamento em relação ao tempo	Que restrições de tempo relacionadas aos processos e funções do software existem?
	Utilização de recursos	Existem restrições de espaço de armazenamento de dados? Limitações de energia?
	Capacidade	Em relação à capacidade de processamento, quais são os valores nominais esperados e quais os valores críticos? Por exemplo, o software pode ser projetado para suportar até 2.000 acessos simultâneos, mas continuaria funcionando mesmo com 10.000 acessos simultâneos.
Segurança (<i>security</i>)	Confidencialidade	Em que grau as informações e funções do sistema devem estar disponíveis apenas para aqueles autorizados a acessá-las?
	Integridade	Em que grau os dados e funções devem ser protegidos contra modificação não autorizada por pessoas ou sistemas?
	Não repúdio	De que forma o sistema deve garantir que os registros mantidos por ele sejam efetivamente verdadeiros de forma que seus autores não possam negar tê-los feito?
	Rastreabilidade de uso	Em que grau as ações do usuário devem ser registradas pelo sistema? Quais informações são mantidas?
	Autenticidade	Em que grau deve-se garantir que um usuário logado seja realmente quem ele diz ser?
Compatibilidade (<i>compatibility</i>)	Coexistência	Com quais produtos o software deve potencialmente coexistir? Quais ferramentas e linguagens de programação devem ser usadas? É necessário rodar o sistema junto a sistemas legados?
	Interoperabilidade	Quais outros produtos devem se comunicar com o sistema? Para quais outros sistemas ele deve enviar dados? De quais sistemas ele deve receber dados?

Tabela 3.1 Classificação de requisitos suplementares baseada na ISO/IEC 25010:2011 e questões geradoras de requisitos.

Característica 25010	Subcaracterística 25010	Questões geradoras de requisitos
Manutenibilidade (<i>maintainability</i>)	Modularidade	Que tipo de arquitetura será usada? Camadas? Partições? Componentes? Serviços Web?
	Reusabilidade	O sistema será produzido com partes de outros sistemas? O sistema deve ser projetado de forma que suas partes possam ser reusadas depois?
	Analisabilidade	Técnicas ou ferramentas especiais serão usadas para facilitar a depuração do código?
	Modificabilidade	Técnicas ou ferramentas especiais serão usadas para garantir que mudanças introduzidas no sistema não produzam novos defeitos?
	Testabilidade	Técnicas ou ferramentas especiais serão usadas para facilitar os testes de regressão? Ferramentas de teste automatizados serão usadas? <i>Test-Driven Development</i> (TDD) será usado? Como o patrimônio de testes produzido durante o desenvolvimento será mantido?
Portabilidade (<i>portability</i>)	Adaptabilidade	Em que grau o software deve adaptar-se a outros contextos além daqueles para os quais ele foi originalmente projetado? Ele deve ser recompilado ou apenas reconfigurado? O que pode ser configurado no sistema? Exemplos de itens configuráveis são impressoras, moeda, fontes, linguagem etc. Requisitos transitórios estarão provavelmente relacionados a itens configuráveis.
	Instalabilidade	Que recursos de instalação devem ser disponibilizados? A instalação deve ser automática? A migração de dados deve ser automática?
	Substituibilidade	Que recursos o sistema deve fornecer quando ele for usado para substituir outros sistemas com os mesmos objetivos? Que recursos o sistema deve fornecer quando ele for substituído por outros sistemas com os mesmos objetivos? Ele deve gerar dados de configuração universalmente compreendidos como XML?
Efetividade (<i>effectiveness</i>)	Efetividade	Que objetivos de negócio no ambiente real de uso o sistema deve ajudar a obter? Em que grau o sistema deve ser responsável por obter esses objetivos de forma completa e correta?
Eficiência (<i>efficiency</i>)	Eficiência	Que tipo de retorno de investimento (ROI) o sistema deve produzir para o cliente?
Satisfação (<i>satisfaction</i>)	Utilidade	De que forma o software deve ser projetado para ajudar o usuário a perceber as consequências do seu uso?
	Prazer	Em que grau o sistema deve proporcionar prazer aos seus usuários?
	Conforto	Em que grau o sistema deve preservar ou melhorar o conforto físico e mental dos usuários?
	Confiança	De que forma o sistema deve ser projetado para que os interessados confiem que ele fará o trabalho?
Uso sem riscos (<i>freedom from risk</i>)	Mitigação de risco econômico	Que tipo de riscos financeiros (incluindo danos morais e materiais) o sistema deve reduzir?
	Mitigação de risco a saúde e segurança	Que tipo de riscos físicos às pessoas o sistema deve reduzir?
	Mitigação de risco ambiental	Que tipo de risco ambiental e à propriedade o sistema deve reduzir?
Cobertura de contexto (<i>context coverage</i>)	Compleitude de contexto	Em que grau o sistema deve ser usado efetiva e eficientemente, sem riscos e com satisfação do usuário em seus contextos de uso? Quais são estes contextos? Existem requisitos legais relacionados com o uso do software?
	Flexibilidade	Em que grau o sistema deve ser usado efetiva e eficientemente, sem riscos e com satisfação do usuário em contextos diferentes daqueles para os quais ele foi originalmente projetado? Quais são estes contextos?

Embora esta lista seja extensa, a equipe deve ter em mente que ela é apenas uma classificação para melhorar a sua capacidade de identificar requisitos que sejam realmente importantes. Não há necessidade de se procurar por requisitos inexistentes, por exemplo, estabelecendo requisitos complicados de empacotamento para um cliente que não se importa com a forma como o software será empacotado.

Também não é recomendável perder muito tempo discutindo se um dado requisito pertence a este tipo ou àquele. Mais importante do que decidir a qual tipo o requisito pertence é saber que ele existe: longas discussões sobre classificação de requisitos não adicionam conhecimento significativo ao projeto.

A Tabela 3.2 apresenta um exemplo de requisitos suplementares que poderiam ser atribuídos ao sistema Livir.

Tabela 3.2 Requisitos suplementares para o sistema Livir.		
Característica 25010	Subcaracterística 25010	Questões geradoras de requisitos
Adequação funcional (<i>functional suitability</i>)	Completude funcional	
	Funcionalidade apropriada	
	Corretude funcional (acurácia)	
Confiabilidade (<i>reliability</i>)	Maturidade	
	Disponibilidade	O sistema deve estar disponível continuamente (24 horas por dia e 7 dias por semana).
	Tolerância a falhas	Transações parcialmente concluídas pelo usuário, no caso de falhas temporárias, devem ser preservadas para recuperação por pelo menos 24 horas. Após este prazo, elas podem ser removidas.
	Recuperabilidade	No caso de falha de energia, comunicação ou dados, o sistema deve tornar-se operacional novamente sem intervenção do operador.
Usabilidade (<i>usability</i>)	Apropriação reconhecível	Todas as funções que geram valor para cada tipo de usuário devem ser facilmente reconhecíveis na página inicial do sistema. Funções primordiais devem ser destacadas.
	Inteligibilidade	Nome de funções (por exemplo, botões e menus) devem se conformar aos nomes usuais do Windows para melhorar a habilidade dos usuários de aprender seu uso.
	Operabilidade	O uso do sistema deve ser claro para usuários novos: eles não devem depender de ajuda ou explicação suplementar.
	Proteção contra erro de usuário	A aplicação deve evitar que o usuário entre com informação inconsistente.
	Estética de interface com usuário	
	Acessibilidade	O sistema deve ser configurável para várias línguas.
Eficiência de desempenho (<i>performance efficiency</i>)	Comportamento em relação ao tempo	Durante as horas de pico é importante que o desempenho do sistema não degrade.
	Utilização de recursos	
	Capacidade	Picos de 10.000 acessos simultâneos são esperados.

Tabela 3.2 Requisitos suplementares para o sistema Livir.

Característica 25010	Subcaracterística 25010	Questões geradoras de requisitos
Segurança (<i>security</i>)	Confidencialidade	Dados sobre vendas e compradores só são acessíveis pelo próprio comprador, pelo gerente de vendas e funcionários autorizados por ele.
	Integridade	Um comprador só pode alterar seus próprios dados e os dados de pedidos em aberto. Todos os outros dados só podem ser alterados por funcionários autorizados. Níveis diferentes de autorização podem ser definidos.
	Não repúdio	O sistema deve armazenar em um registro à prova de fraude os seguintes dados sobre cada pedido: seu conteúdo, data e hora em que foi emitido e recebido e a identidade do comprador.
	Rastreabilidade de uso	
	Autenticidade	As senhas dos usuários jamais devem ser visíveis, nem mesmo pelos administradores do sistema.
Compatibilidade (<i>compatibility</i>)	Coexistência	
	Interoperabilidade	O sistema deve ser capaz de se comunicar automaticamente com operadoras de cartão de crédito para permitir a confirmação do pagamento.
Manutenibilidade (<i>maintainability</i>)	Modularidade	
	Reusabilidade	
	Analísabilidade	
	Modificabilidade	
	Testabilidade	
Portabilidade (<i>portability</i>)	Adaptabilidade	O servidor deve ser instalado em uma única plataforma. A aplicação cliente deve ser acessível pelo menos pelo Internet Explorer, Google Chrome e Mozilla Firefox. A aplicação cliente deve ser sensível à configuração do computador do usuário para identificar a língua, moeda e outras definições.
	Instalabilidade	O comprador pode acessar o sistema via Web sem a necessidade de instalar qualquer componente adicional, exceto aqueles que usualmente estão disponíveis com os browsers.
	Substituibilidade	
Efetividade (<i>effectiveness</i>)	Efetividade	
Eficiência (<i>efficiency</i>)	Eficiência	
Satisfação (<i>satisfaction</i>)	Utilidade	
	Prazer	
	Conforto	
	Confiança	
Uso sem riscos (<i>freedom from risk</i>)	Mitigação de risco econômico	
	Mitigação de risco a saúde e segurança	
	Mitigação de risco ambiental	

Tabela 3.2 Requisitos suplementares para o sistema Livir.

Característica 25010	Subcaracterística 25010	Questões geradoras de requisitos
Cobertura de contexto (<i>context coverage</i>)	Completude de contexto	
	Flexibilidade	

Nem todos os campos foram preenchidos porque, como dito antes, requisitos não devem ser inventados, devem ser solicitados pelo cliente. Assim, usualmente não há requisitos em todas as categorias. As questões geradoras de requisitos mencionadas na Tabela 3.1 são uma boa base para encontrar necessidades eventuais, mas não é obrigatório ter uma resposta para todas aquelas questões.

3.6 Modelo conceitual preliminar

Embora os Capítulos 6 e 7 apresentem técnicas de modelagem conceitual em detalhes, é necessário mencionar aqui que existem relações de interdependência entre os casos de uso de sistema e o chamado *modelo conceitual preliminar* (Larman, 2004). O modelo conceitual preliminar é construído durante a Concepção e consiste em um diagrama que representa as principais unidades de informação do sistema. Ainda não é necessário representar atributos. Embora as associações possam aparecer no modelo, ainda não é necessário também detalhar suas características.

Com a análise do diagrama de casos de uso de sistema, vários conceitos importantes podem ser descobertos. Esses conceitos são representados como classes no modelo conceitual preliminar: eles representam a estrutura da informação que será gerenciada pelo sistema. Ao mesmo tempo, um analista que observe o modelo conceitual poderá perceber se o diagrama de casos de uso é suficientemente completo. Essa verificação usualmente ocorre quando novas entidades são identificadas em um processo de negócio e se torna necessário que elas sejam de alguma forma registradas pelo sistema.

A Figura 3.9 apresenta um possível modelo conceitual preliminar para o diagrama de casos de uso apresentado na Figura 3.6. O processo de descobrir classes consiste em pensar sobre os casos de uso e imaginar quais informações (de alto nível) devem ser trocadas entre os atores e o sistema para que o processo funcione. As associações entre classes representam dependências ou relacionamentos entre as peças de informação.

Ao olhar para os casos de uso da Figura 3.6, com o objetivo de encontrar os conceitos da Figura 3.9, pode-se ver que:

- *Pedir livros* é provavelmente o caso de uso mais importante do sistema. Ele dá origem a dois conceitos, *Livro* e *Pedido*, os quais são associados.

- *Pagar livro* adiciona um novo conceito, *Pagamento*, o qual é associado com o *Pedido*. Neste ponto, a equipe poderia decidir trocar o nome deste caso de uso para *Pagar pedido*.
- O caso de uso *Receber livros* dá origem ao conceito *Chegada*, que é associado a *Livro*. Possivelmente, neste ponto, *Pedido De Compra* poderia ser adicionado como um novo conceito.

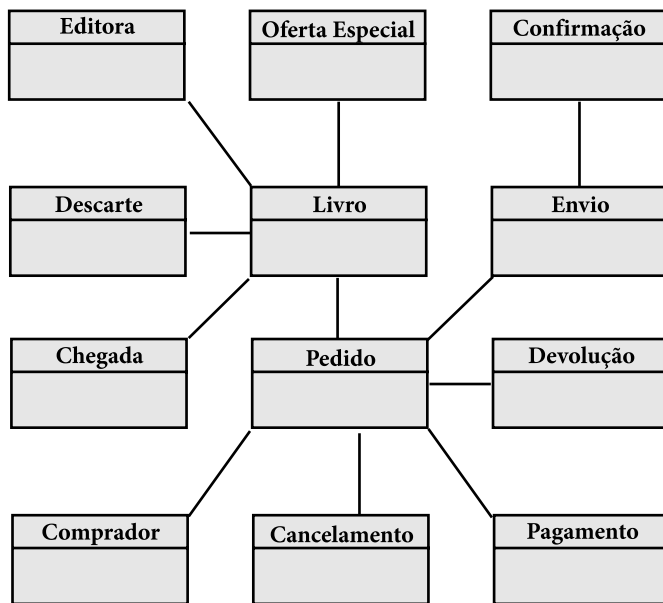
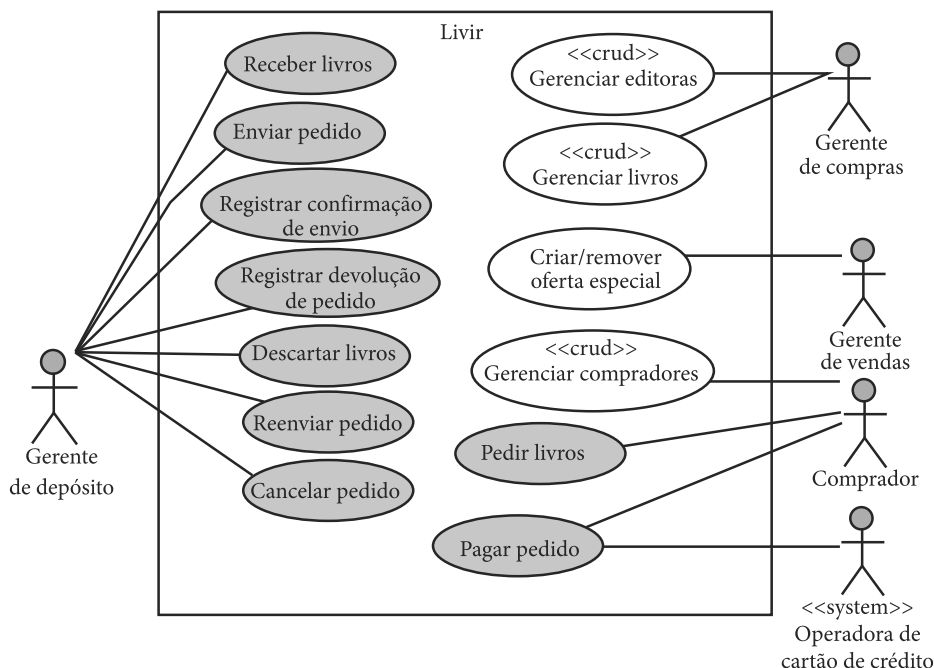


Figura 3.9

Modelo conceitual preliminar para os casos de uso da Figura 3.6.

- *Enviar livros* produz um novo conceito, *Envio*, que é associado com *Pedido*. Agora, *Enviar livros* poderia ser renomeado para *Enviar pedido*.
- *Registrar confirmação de entrega* produz o conceito *Confirmação*, que é associado com *Envio*.
- *Registrar devolução de livros* produz *Devolução*, que é associada a *Pedido*. Talvez, neste ponto, o caso de uso seja renomeado para *Registrar devolução de pedido*.
- *Descartar livro* produz *Descarte*, que é associado a *Livro*.
- *Reenviar livros* não produz nenhum conceito novo à primeira vista porque pode ser considerado uma repetição do caso de uso *Enviar livros*. Entretanto, isso precisaria ser analisado melhor mais adiante.
- *Cancelar venda* produz o conceito *Cancelamento*, que é associado a *Pedido*. Talvez o caso de uso seja renomeado para *Cancelar pedido* neste ponto.
- *Criar/remover oferta especial* produz o conceito *Oferta Especial*, que é associado a *Livro*.

**Figura 3.10**

Modelo de casos de uso com nomes revisados e CRUDs adicionados.

A identificação do modelo conceitual preliminar é especialmente útil para facilitar a visualização da estrutura da informação que vai ser gerenciada pelo sistema; isso ajuda a unificar o vocabulário entre os membros da equipe e outros interessados. As decisões sobre trocar nomes de casos de uso para limpar o vocabulário já são implementadas na Figura 3.10.

Existe ainda mais uma utilidade prática importante para o modelo conceitual preliminar. Entre os conceitos mostrados, a maior parte é de elementos de informação criados ou alterados no contexto dos casos de uso já identificados. Entretanto, alguns destes conceitos não são nem criados nem alterados nestes casos de uso e isso significa que alguns casos de uso ainda podem estar faltando. Este é o caso especialmente das classes *Livro*, *Editora* e *Cliente*. Esses conceitos podem ser considerados CRUD, porque permitem as quatro operações clássicas: *Create*, *Retrieve*, *Update* e *Delete*⁷. Se elas vão ser adicionadas ao diagrama, em vez de representá-las individualmente, será melhor representar as quatro operações como um único caso de uso CRUD, que é estereotipado com <<crud>>, como mostrado na Figura 3.10.

Um novo ator foi definido para gerenciar as editoras e livros: o *Gerente de compras*. No entanto, foi decidido que o comprador poderia gerenciar seus próprios dados.

⁷ Inserir, consultar, atualizar e remover, respectivamente.

Por que não há CRUDs para pedidos, cancelamentos, devoluções, pagamentos etc.? É porque estes elementos já são gerenciados pelos casos de uso que já foram identificados no diagrama e não é necessário criar outros casos de uso especificamente para eles. Por exemplo, um pedido é criado pelo caso de uso *Pedir livros*; ele é modificado por casos de uso como *Cancelar pedido* e *Pagar pedido*; é visualizado (consultado) em vários casos de uso e, finalmente, assume-se que ele não possa ser simplesmente removido do sistema.

Outra questão que poderia ser levantada com os interessados é: quais são os relatórios que o sistema deve produzir? Embora eles sejam os casos de uso mais triviais, como explicado depois, relatórios são uma importante fonte para identificar as informações necessárias para satisfazer os interessados.

Há uma diferença entre estes relatórios e a consulta CRUD: a consulta consiste simplesmente em recuperar dados armazenados sobre um único objeto; um relatório, no entanto, usualmente envolve um número maior de objetos, muitas vezes até de classes diferentes, e necessariamente inclui algum tipo de filtro ou combinação de dados (soma, produto, média, maior valor, menor valor etc.). Por exemplo, uma consulta por informações sobre um livro pelo seu ISBN não deveria ser considerada um relatório porque ela já está incluída no caso de uso CRUD.

Para o exemplo corrente, alguns relatórios que são interessantes para certos atores poderiam ser identificados, como:

- *Gerente de depósito*: relatório de pedidos por período de tempo, relatório de envios por período de tempo, relatório sobre o total de livros disponíveis para venda, relatório de devoluções por período de tempo e relatório de livros descartados por período de tempo.
- *Comprador*: relatório de status de pedido e relatório de vendas passadas.
- *Gerente de vendas*: relatório de vendas por período de tempo.

O cliente poderia solicitar outros relatórios, pois estes anteriormente mostrados são apenas exemplos. Aqueles que são usados exclusivamente dentro de um caso de uso existente não devem ser incluídos nessa relação. Por exemplo, o gerente de depósito poderia estar interessado em listar os pedidos de compra pendentes. Porém, se esta lista somente será consultada no momento em que um pedido de compra chegar ao depósito e precisar ser registrado, então a consulta aos pedidos pendentes deve ser considerada parte do caso de uso *Receber livros* e não deve ser incluída no diagrama. Apenas relatórios que não são necessariamente parte de outros casos de uso devem ser incluídos no diagrama: caso contrário, a lista poderá rapidamente crescer tanto a ponto de não ser mais gerenciável e, como consequência, ficar redundante ou incompleta (porque será a expansão dos casos de uso com diagramas de sequência que realmente vai indicar

quais passos, incluindo consultas, são necessários para cada caso de uso). A Figura 3.11 mostra o diagrama de casos de uso atualizado com os relatórios (estereotipados com `<<report>>`), como indicado anteriormente.

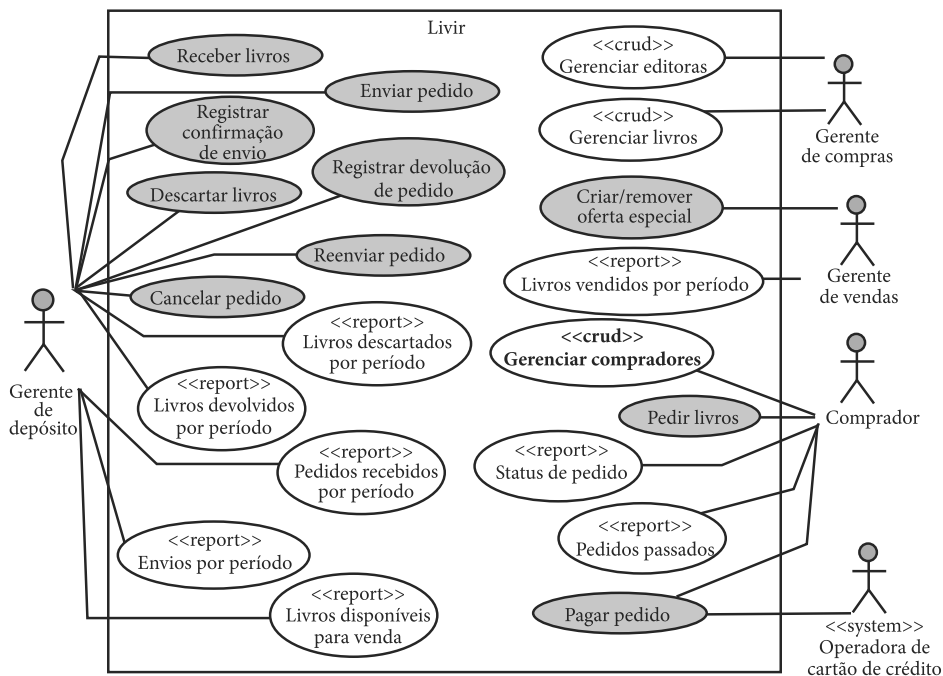


Figura 3.11

Diagrama de casos de uso com relatórios⁸.

A quantidade de casos de uso `<<report>>` dependerá da estrutura da informação que eles apresentam. Quando a parametrização for possível, ela deve ser usada. Não é necessário, por exemplo, ter um relatório de vendas semanal e um relatório de vendas mensal. A não ser que tenham uma estrutura de dados distinta, eles devem ser considerados um único relatório. A referência a semana, mês ou qualquer outro período seria simplesmente um parâmetro.

Entretanto, não é recomendável agrupar relatórios de naturezas diversas, como, por exemplo, *Relatório de vendas por livro* e *Relatório de envios por período*: os parâmetros e saídas são diferentes. Assim, estes dois relatórios devem ser considerados dois casos de uso distintos.

Como visto na Figura 3.11, o número de casos de uso pode se tornar grande e o diagrama rapidamente ficar difícil de organizar. Esta é uma das razões para evitar a inclusão de fragmentos nesses diagramas. Cada caso de uso individual será detalhado por outros

⁸ Casos de uso estereotipados com `<<report>>` têm seus nomes abreviados para evitar a repetição da expressão "Gerar relatório para..." que seria usada em cada um deles.

meios. Além disso, pode-se escolher não incluir os casos de uso CRUD e relatórios no diagrama no caso de sistemas de tamanho moderado a grande. Eles podem ser listados separadamente (em uma planilha ou um diagrama de casos de uso separado, por exemplo), de forma a não poluir o diagrama de casos de uso principal. Outra solução que pode ser usada em alguns casos é representá-los em uma cor diferente (como feito nas Figuras 3.10 e 3.11) de forma que eles não dificultem a visualização dos casos de uso-chave no diagrama.

3.7 O processo visto aqui

	Concepção	Elaboração
Modelagem de negócio		
Requisitos	Preparar o diagrama de casos de uso de sistema (requisitos funcionais): • Identificar os atores de sistema a partir do modelo de casos de uso de negócio. • Identificar os casos de uso de sistema a partir do modelo de casos de uso de negócio e dos diagramas de atividades e de máquina de estado do modelo de negócio. Identificar os requisitos não funcionais como anotações aos casos de uso: • Identificar as principais regras de negócio associadas aos casos de uso. • Identificar os principais aspectos de qualidade relacionados aos casos de uso. • Identificar requisitos suplementares.	
Análise e design	Preparar o modelo conceitual preliminar a partir da observação dos casos de uso de sistema e dos conceitos que eles necessitam.	
Implementação		
Teste		
Gerenciamento de projeto		

3.8 Questões

1. Explique as diferenças entre um caso de uso de negócio e um caso de uso de sistema.
2. Qual a utilidade de um caso de uso de sistema ao longo do processo de desenvolvimento de software?
3. Quais atores e trabalhadores de negócio são convertidos em atores de sistema?
4. O que são requisitos funcionais, não funcionais e suplementares e quais características eles podem ter?
5. Por que um modelo conceitual preliminar deve ser feito durante a fase de Concepção?

Planejamento de projeto baseado em casos de uso

TÓPICOS-CHAVE NESTE CAPÍTULO

- Estimção de esforço
- Gerenciamento de risco
- Pontos de caso de uso
- Planejamento de projeto com iterações

4.1 Introdução à estimção de esforço e análise de risco em projetos de software

A motivação para estimção de esforço em projetos de software vem do fato de que historicamente a maioria desses projetos:

- Demora mais tempo para ser completada do que o esperado.
- Custa muito mais do que o esperado.
- Gera um produto que não tem a qualidade desejada.
- Gera um produto que não tem o escopo desejado.

Assim, a questão é que equipes de desenvolvimento de software e clientes devem aprender a esperar o tempo, custo, qualidade e escopo corretos em relação ao seu projeto e sua equipe. Técnicas de estimção de esforço são usadas para alcançar estes objetivos.

4.1.1 Técnicas *ad hoc*

Várias técnicas de estimção de esforço foram propostas e usadas nas últimas décadas. Uma que é muito popular é a técnica *João* de que pode ser resumida assim: “João, de quanto tempo você precisa para desenvolver este sistema?”. Uma variação da técnica João é a técnica *João com restrições*, que pode ser resumida como “João, você tem três meses para desenvolver este sistema!”. Embora popular (e conhecida por outros nomes como Maria, Pedro, Luiza etc.), esta abordagem não é muito precisa.

Outra técnica *ad hoc* usada é a dos *Seis Meses*. Ela consiste em estimar “seis meses” para qualquer projeto de software no seu início e então ir ajustando para cima ou para

baixo à medida que o escopo ou requisitos são descobertos. Variações dessa técnica são virtualmente infinitas, incluindo a técnica *Um Ano* e a técnica *Dezoito Meses*, entre outras.

As técnicas deste tipo são conhecidas como *estimação não paramétrica* porque elas não são baseadas em medidas do projeto a ser desenvolvido. Somerville (2006) identifica algumas outras técnicas não paramétricas:

- *Opinião de especialista.* A técnica de buscar a opinião de especialistas propõe que um ou mais especialistas no domínio do projeto e em desenvolvimento de software devem se encontrar e usar sua experiência para produzir uma estimativa. A técnica João é um cenário potencialmente ruim para a técnica de opinião de especialista: a técnica pode ser muito pouco precisa, dependendo da experiência dos estimadores. Entretanto, se os especialistas realmente possuem experiência em projetos similares, ela é uma técnica factível, rápida e relativamente barata.
- *Estimação por analogia.* Esta técnica é de fato a base pragmática para a opinião de especialista. Assume-se que o esforço para desenvolver um novo sistema deve ser similar àquele que foi despendido para desenvolver sistemas semelhantes. A técnica não será viável se não houver informações sobre projetos similares para comparação. Ela pode fazer bom uso de um ou mais especialistas para determinar o que qualifica um sistema como similar e como o tempo que foi gasto para desenvolvê-lo pode ser usado como base para estimar o esforço para o projeto atual. Assim, esta técnica usualmente envolve a opinião de especialistas.
- *Lei de Parkinson.* Esta técnica, de maneira geral, não é adotada abertamente, mas ela é recorrente em projetos de software e em outras áreas: *o projeto vai custar a quantidade de recursos que estiver disponível*. A vantagem é que o projeto não vai custar mais do que o esperado, mas frequentemente o escopo poderá estar aquém do esperado.
- *Preço para vencer.* O custo de um projeto é igual ao preço que se acredita vencerá o contrato. A desvantagem deste método é que o sistema que o cliente obtém pode não ser o que ele espera, porque, como os custos são usualmente reduzidos, isso pode impactar a qualidade e o escopo. Entretanto, se não houver informação detalhada sobre o sistema e a equipe não tiver experiência com técnicas de estimação mais avançadas, esta técnica pode ser a escolhida: o custo do projeto será acordado baseado em entendimentos e o desenvolvimento será restringido pelo custo.

A comunidade ágil usualmente adota uma estratégia de estimação baseada em *pontos de história*. Uma *história de usuário* é um cenário de uso de um sistema e é, de certa forma, similar a um caso de uso. Entretanto, casos de uso são mais estruturados e formais; eles são produzidos pela equipe de analistas. Histórias de usuário, no entanto, devem ser escritas pelos próprios usuários. Elas devem representar um cenário onde os usuários veem a si próprios usando o sistema que vai ser produzido. A ideia por trás disso é que os usuários apresentem as necessidades que são mais importantes para eles primeiro; os detalhes serão lembrados depois.

Um *ponto de história* é considerado um *dia ideal de trabalho* (6 a 8 horas focadas e dedicadas a um projeto). A estimativa de esforço é então calculada para cada história de usuário em termos de pontos de história. A questão a ser respondida pela equipe é “Quantos dias ideais x pessoas levariam para desenvolver uma dada história de usuário?”. Se a resposta for “ x pessoas levariam y dias”, então o número de pontos de história é $x \cdot y$. A atribuição de pontos de história é subjetiva e pode ser considerada uma técnica não paramétrica. A equipe usualmente avalia o esforço baseado em sua experiência em projetos passados similares. A atribuição é baseada em esforço, complexidade e risco. Por exemplo, as seguintes questões poderiam ser levantadas:

- *Complexidade*: esta história de usuário tem quantos cenários possíveis?
- *Esforço*: a mudança terá de ser feita em muitos componentes diferentes?
- *Risco*: alguém na equipe conhece a tecnologia necessária para desenvolver esta história?

Outra forma de estimar histórias de usuário é colocar todas as histórias em cartões sobre a mesa. Então, as mais simples de desenvolver são separadas e um ponto de história ou menos é atribuído a cada uma. Depois disso, as mais simples entre as restantes são separadas e dois pontos de história atribuídos a cada uma. Este procedimento é repetido até que não restem mais histórias sobre a mesa. É sugerido que cada iteração aumente o número de pontos de história seguindo uma série similar a uma sequência de Fibonacci aproximada, como, por exemplo, 0, $\frac{1}{2}$, 1, 2, 3, 5, 8, 13, 20, 40, 100 etc.¹. A razão por trás disso é que para um ser humano comum é difícil estimar quanto um cavalo pesa; entretanto, a maioria das pessoas vai concordar que um cavalo pesa mais que um cão e menos do que um elefante.

Se uma história de usuário é estimada em menos de um dia ideal de trabalho, ela pode ser agrupada com outra história, e histórias acima de um dado limite (por exemplo, 10 pontos de história) deveriam ser divididas em histórias mais simples.

4.1.2 Técnicas paramétricas

Outra classe de técnicas de estimativa inclui as paramétricas, que procuram apresentar uma estimativa de esforço baseada no tamanho estimado do sistema, na dificuldade técnica de desenvolvê-lo e na capacidade da equipe, entre outras características. O tamanho de um sistema pode ser medido em termos de:

- *Linhas de código*. Técnicas como COCOMO II – CONstrutive COst MOdel II (Boehm, 2000) iniciam estimando quantas *linhas de código fonte* (Source Lines Of Code – SLOC) o sistema terá. De fato, como a maioria dos sistemas tem milhares de linhas de código, a medida usual é dada em termos KSLOC, uma unidade que representaria cerca de mil linhas de código.

¹ Na verdadeira série de Fibonacci, cada número é a soma dos dois anteriores. Assim, a série original é 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 etc. Entretanto, para o propósito de estimativa, uma série aproximada é usada porque, à medida que os números crescem, a estimativa se torna menos precisa.

- *Pontos de função.* Técnicas baseadas em *análise de pontos de função* (Function Points Analysis – FPA) (Albrecht, 1979) e suas variações não se preocupam com o número de linhas de código. Elas estimam que o esforço para desenvolver um sistema depende da *funcionalidade aparente* que vai ser implementada. Assim, usualmente, essas técnicas contam o número de requisitos funcionais e avaliam sua complexidade perguntando quantas classes ou registros de dados devem ser usados para implementar a função e quantos argumentos um usuário vai enviar ou receber do sistema para realizar a função.
- *Pontos de caso de uso.* É uma variação da técnica de análise de pontos de função que se baseia em casos de uso em vez de funções individuais para fazer a estimação. Essa técnica é explicada em detalhes na Seção 4.2.

Embora COCOMO II seja a técnica mais detalhada, ela tem uma desvantagem quando comparada com pontos de função e pontos de caso de uso: o número de KSLOC precisa ser estimado por especialistas antes da aplicação do método e esta é uma fonte significativa de incerteza². Pontos de função e pontos de caso de uso são técnicas baseadas em requisitos, os quais podem ser realmente conhecidos no início de um projeto (é claro que requisitos podem mudar durante o desenvolvimento, mas se isso acontecer as estimativas podem ser imediatamente ajustadas de forma sistemática).

Em adição ao tamanho do sistema medido em linhas de código ou funcionalidade aparente, as técnicas paramétricas também fazem uso de fatores de *ajuste técnico*. Assume-se que implementar funções com o mesmo tamanho pode ser mais fácil ou mais difícil, dependendo do tipo de sistema que está sendo desenvolvido. Por exemplo, um sistema de informação como Livir é mais fácil de implementar do que um sistema que controle um robô cirurgião ou um sistema embarcado que controle as funções de um avião a jato. A mesma quantidade de linhas de código pode levar mais ou menos tempo para ser desenvolvida dependendo da complexidade técnica do sistema.

A técnica mais completa em relação ao ajuste dos fatores técnicos é COCOMO II, que propõe o uso de até 16 fatores multiplicadores de esforço divididos em quatro grupos:

- *Atributos do produto.* Incluem a confiabilidade requerida para o software, o tamanho relativo do banco de dados de testes, a complexidade interna do produto, o desenvolvimento visando reusabilidade e a razão entre a quantidade de documentação esperada e necessária.
- *Atributos do hardware.* Incluem aspectos relacionados à eficiência de tempo e espaço e à volatilidade da plataforma de desenvolvimento.
- *Atributos do pessoal.* Incluem a capacidade dos analistas e programadores, continuidade do pessoal, experiência em aplicações similares e experiência na plataforma, linguagem de programação e outras ferramentas de software.

² Para minimizar esta incerteza, tabelas *backfire* que convertem pontos de função em KSLOC podem ser usadas (Boehm, 2000).

- *Atributos do projeto.* Estes incluem o uso de ferramentas CASE, a distribuição física da equipe de desenvolvimento e a necessidade de acelerar o cronograma de desenvolvimento.

Pontos de caso de uso e pontos de função têm multiplicadores de esforço similares, mas uma organização e muitas vezes uma interpretação diferente. Em todos os métodos, os multiplicadores de esforço consistem em um índice numérico que é multiplicado pelo tamanho do sistema para produzir uma estimação de esforço. Entretanto, COCOMO II tem uma característica única: um conjunto de cinco *fatores de escala*. São eles:

- *Precedentes.* O produto é similar a outros que já foram desenvolvidos pela mesma equipe?
- *Flexibilidade no desenvolvimento.* O produto deve ser desenvolvido estritamente de acordo com os requisitos ou os requisitos são apenas objetivos genéricos?
- *Resolução de riscos e arquitetura.* Existe um bom suporte para resolver riscos e questões arquiteturais?
- *Coesão da equipe.* A equipe já trabalhou junta antes? Seus membros estão integrados?
- *Maturidade de processo.* Qual a maturidade da organização? CMMI³ (SEI-CMU, 2010) e SPICE⁴ (Emam; Melo; Drouin, 1997) podem ser usadas como medida de maturidade. COCOMO II também propõe um questionário simples que ajuda a estimar a maturidade de uma organização se outras avaliações não estiverem disponíveis.

A diferença aqui é que esses fatores de escala não são multiplicados pelo tamanho do sistema: eles são usados em uma *exponencial*. Valores médios (próximos de 1) para esses fatores significam uma escala nominal onde cada linha de código produzida custa o mesmo independentemente do tamanho do software. Se os fatores de escala são ruins, então seu valor numérico será maior do que 1 e isso significa que uma linha de código em um sistema maior custará mais caro que uma linha de código em um sistema menor. Se os fatores de escala são bons, então a equipe ganha em escala, isto é, uma linha de código em um sistema maior custa menos que uma linha de código em um sistema menor.

Maiores informações sobre estimação de esforço e outras técnicas de planejamento e gerenciamento de projetos de software podem ser encontradas em Wazlawick (2013).

4.1.3 Análise de riscos

Um *risco* é algo que pode acontecer e causar problemas⁵. Riscos são muito frequentes e numerosos em projetos de software. Existem diferentes categorias de risco e essas podem ter diferentes impactos: há alguns riscos que apenas causam desconforto e outros que podem causar o cancelamento do projeto. Usualmente, o processo de lidar com riscos envolve:

³ *Capability Maturity Model Intergration.*

⁴ *Software Process Improvement and Capability Determination.*

⁵ Algumas definições também consideram riscos *positivos* situações que podem acontecer e ajudar o projeto.

- *Identificação.* A equipe deve descobrir e conhecer as condições que podem iniciar problemas em um projeto. *Checklists* como a de Carr *et al.* (1993) podem ajudar no processo de identificação.
- *Análise.* Riscos são analisados de forma que alguns de seus atributos sejam descobertos. Usualmente, pelo menos dois atributos devem ser identificados: a *probabilidade* de que a condição que transforma o risco em problema realmente ocorra e o *impacto* que o risco pode causar no projeto se ocorrer. A *exposição* ou *importância* de um risco é uma combinação da sua probabilidade e seu impacto. Usualmente, riscos com alta probabilidade de ocorrência e alto impacto no projeto são os mais importantes.
- *Planejamento.* Uma vez que a exposição dos riscos tenha sido avaliada, planos devem ser feitos para riscos de alta e média exposição e o projeto deve incluir atividades para minimizar a probabilidade e/ou o impacto dos riscos mais importantes. Esses planos são chamados de *planos de mitigação*. Além disso, *planos de recuperação* ou *contin-gência* também devem ser preparados para os riscos mais importantes, e eles serão executados se, apesar das atividades de mitigação, o risco se tornar um problema.
- *Controle.* Os riscos de um projeto são ainda mais instáveis do que seus requisitos: eles mudam frequente e inesperadamente. Novos riscos são descobertos durante o desenvolvimento e sua exposição pode mudar. Assim, o gerente de projeto deve estar atento e prestar grande atenção ao *status* dos riscos durante o projeto inteiro.

Diferentes fontes de riscos podem ser identificadas. Carr *et al.* (1993) apresentam a seguinte estrutura:

- *Engenharia de produto:*
 - *Requisitos:* eles são estáveis, completos, claros, válidos e factíveis? O projeto é algo que nunca foi feito antes? O projeto é maior do que outros que a organização já fez antes?
 - *Design:* ele é difícil de se obter? As interfaces com outros sistemas são claras e fáceis? Existem restrições de desempenho, testabilidade e hardware? Software desenvolvido fora da empresa será usado?
 - *Codificação e teste:* a implementação é factível? O tempo reservado para o teste é suficiente?
 - *Integração e teste:* os ambientes de integração e teste são adequados?
 - *Aspectos de engenharia:* o software será fácil de manter? Os requisitos de confiabilidade e segurança são fáceis de obter e verificar? Existem fatores humanos especiais que devem ser levados em conta? A documentação é adequada para o projeto?
- *Ambiente de desenvolvimento:*
 - *Processo de desenvolvimento:* Há um processo claro e provado em uso? Os planos são formalmente controlados? O processo é adequado para este tipo de sistema? Os membros da equipe conhecem o processo? Existem mecanismos claros para gerenciar mudanças no produto?

- *Sistema de desenvolvimento*: Existem hardware adequado e outras ferramentas para o desenvolvimento do projeto? As ferramentas são fáceis de usar? A equipe sabe como usar as ferramentas? As ferramentas são confiáveis?
- *Processo de gerenciamento*: Os planos são adequados? Eles incluem planos de contingência? Os papéis e autoridade dentro do projeto são bem definidos? Os gerentes têm experiência? Como a equipe se comunica com os interessados?
- *Método de gerenciamento*: Existem métricas para ajudar o gerenciamento? O pessoal do projeto é treinado? O controle de qualidade é adequado? A configuração do código e outros artefatos são controlados?
- *Ambiente de trabalho*: A equipe é preocupada com qualidade? A equipe é cooperativa? A comunicação entre os membros da equipe é boa? A moral da equipe é alta?
- *Restrições externas*:
 - *Recursos*: O cronograma é confiável e estável? A equipe é experiente e capaz? O orçamento é adequado? As instalações são adequadas?
 - *Contrato*: que tipo de contrato existe? Existem restrições contratuais? O projeto depende de parceiros externos?
 - *Interfaces de comunicação*: Existem problemas com o cliente, como comunicação fraca ou conhecimento incompleto sobre o domínio?
 - *Pessoal subcontratado*: Existem problemas com empresas parceiras, como interfaces fracas, comunicação pobre ou falta de colaboração? O projeto depende delas?
 - *Principal contratador*: Se você é uma empresa subcontratada, existem dificuldades em relação ao contratador principal?
 - *Gerência*: a alta gerência é ausente ou microgerencia?
 - *Vendedores*: os vendedores de produto que você está usando conseguem resolver problemas rapidamente?
 - *Políticas*: as políticas estão criando problemas para o projeto?

Esta lista é apenas uma referência; muitos outros riscos imprevisíveis podem ser identificados na maioria dos projetos.

Após serem identificados, riscos devem ser analisados. Vários atributos de riscos podem ser identificados. Entre os mais importantes estão:

- *Probabilidade*: a chance de que o risco realmente se torne um problema. Se existir uma série de dados históricos, a probabilidade do risco pode até ser medida numericamente. Entretanto, a maioria dos riscos é avaliada usando-se a escala *camiseta*, em que “grande” significa que é quase certo que o risco vai se tornar um problema; “médio” significa que se espera que o risco possa se tornar um problema eventualmente; e “pequeno” significa que a chance de o risco se tornar um problema é mínima.
- *Impacto*: o preço (monetário ou não) a pagar quando o risco se tornar um problema. Novamente, medidas numéricas podem ser usadas se houver informação suficiente dis-

ponível. Para a maioria dos projetos, o impacto do risco pode ser avaliado como “grande” (pode provocar o cancelamento do projeto); “médio” (pode aumentar significativamente os custos do projeto); e “pequeno” (pode causar transtorno, mas as perdas são facilmente recuperáveis).

- *Importância ou exposição*: a combinação da probabilidade com o impacto, conforme mostrado na Tabela 4.1.

Tabela 4.1 Como calcular a exposição de um risco.				
		Probabilidade		
		Grande	Média	Pequena
Impacto	Grande	Alta exposição	Alta exposição	Média exposição
	Médio	Alta exposição	Média exposição	Pequena exposição
	Pequeno	Média exposição	Pequena exposição	Pequena exposição

Riscos de alta e média exposição devem ter planos de mitigação para reduzir a probabilidade e/ou impacto. No caso de riscos de alta exposição, é aconselhável executar os planos o mais cedo possível, enquanto os riscos de média exposição podem ser tratados mais tarde. Assim, as atividades de mitigação de riscos vão ser incluídas no esforço do projeto neste momento. Riscos de baixa exposição devem ser apenas monitorados para detectar se sua exposição muda durante o projeto.

Entre outras possíveis definições, pode-se dizer que um risco é composto de três partes:

- *Causa*: uma condição incerta que pode criar problemas.
- *Problema*: uma situação que poderia ocorrer dada a incerteza da causa.
- *Efeito*: um impacto sobre um ou mais objetivos do projeto.

Assim, há três tipos de planos que podem ser associados a riscos:

- *Plano de mitigação para reduzir a probabilidade do risco*: ele deve atuar na *causa* do risco, reduzindo a probabilidade de sua ocorrência.
- *Plano de mitigação para reduzir o impacto do risco*: ele deve atuar nos *efeitos* do risco, reduzindo o prejuízo se o risco realmente se tornar um problema.
- *Plano de contingência ou plano de recuperação de desastre*: ele estabelece o que fazer depois que o risco realmente se tornar um problema, ou seja, ele atua no *problema* criado pelo risco.

Planos de mitigação são preventivos: eles deveriam ser executados antes que o risco se tornasse um problema. Planos de contingência, no entanto, são executados somente depois que o risco realmente se tornou um problema.

A Tabela 4.2 apresenta um exemplo com alguns riscos associados ao projeto Livir e algumas simples ações de mitigação e contingência. Planos mais detalhados poderiam ser elaborados, entretanto, se necessário.

Tabela 4.2 Exemplo de riscos e planos para o projeto Livir (P=Probabilidade, I=Impacto e E=Exposição).

Cód.	Risco			Atributos			Plano de redução de probabilidade	Plano de redução de impacto	Plano de contingência
	Causa	Problema	Efeito	P	I	E			
K1	Requisitos instáveis.	Requisitos chave podem mudar durante o desenvolvimento.	Tempo perdido desenvolvendo partes que não serão usadas.	G	G	G	Ter cuidado especial durante a eliciação de requisitos. Estudar produtos similares. Desenvolver protótipos para validação de requisitos.	Enfatizar desenvolvimento modular. Implementar um controle de versões eficiente.	Usar um sistema de gerenciamento de mudança para controlar as mudanças nos requisitos adequadamente.
K2	A equipe não tem experiência.	Decisões de design ruins podem ser tomadas.	Código difícil de manter e cheio de defeitos.	G	M	G	Contratar um arquiteto de software experiente para coordenar e validar todas as decisões de design.	Usar teste automático extensivamente para encontrar defeitos.	Refatorar código para melhorar o design.
K3	A equipe não está familiarizada com o protocolo de comunicação com as operadoras de cartão de crédito.	Interfaces de comunicação erradas ou ineficientes com operadoras de cartão de crédito.	Erros em operações com cartão de crédito	G	P	M	Desenvolver um protótipo descartável para estudar o protocolo de comunicação do sistema da operadora de cartão de crédito.	Implementar procedimentos de dupla verificação para operações de cartão de crédito baseados em invariantes e pós-condições.	Dobrar a verificação de segurança em operações de cartão de crédito. Interromper operações com cartões até que os problemas sejam consertados.

Riscos devem ser monitorados enquanto o projeto estiver em desenvolvimento. Assim como os requisitos, riscos usualmente mudam com o passar do tempo. Mesmo seus nomes e propriedades podem mudar. É por isso que os riscos são identificados por um código: para mantê-los rastreados mesmo se mudarem.

4.2 Análise de pontos de caso de uso⁶

Uma vez que casos de uso de sistema tenham sido identificados, o esforço necessário para desenvolvê-los pode ser estimado. Além disso, os casos de uso devem ser priorizados de forma que os mais importantes sejam desenvolvidos primeiro. Em um processo iterativo como o Processo Unificado, um conjunto relativamente pequeno de casos de uso seria desenvolvido em cada iteração. Assim, planejar um projeto usando este tipo de processo consiste em determinar um plano geral ou *plano de fase* e um conjunto de *planos de iteração* que determinam quais casos de uso ou riscos serão tratados em cada iteração. Por isso é necessário determinar as prioridades de um projeto e estimar o esforço necessário

⁶ Leitores que não estiverem interessados em conhecer detalhes sobre como estimar o esforço e desenvolver um plano de projeto poderão pular daqui diretamente para o Capítulo 5, onde as técnicas de análise e design orientadas a objetos continuam a ser explicadas.

para desenvolvê-lo. Durante a Concepção, apenas o plano de fase e o plano da primeira iteração da Elaboração são construídos. Cada iteração seguinte é planejada enquanto a corrente está sendo realizada.

Planejamento de fase e iteração são atividades típicas de gerenciamento de projeto. Estimação de esforço é uma ferramenta de planejamento que indica, entre outras coisas, quanto tempo e quanto pessoal é necessário para o projeto.

A técnica conhecida como *pontos de caso de uso* (Karner, 1993) é baseada em *pontos de função* (Albrecht; Gaffney Jr., 1983), especificamente na técnica de contagem *MK II* ou *Mark II* (Symons, 1988), que é uma abordagem relativamente mais simples do que a do *International Function Point Users Group* (IFPUG)⁷.

Pontos de casos de uso representam uma técnica mais simples do que análise e pontos de função, mas ela ainda sofre por não ter relatos consistentes de sua aplicação a um volume grande de projetos, ao contrário de técnicas como COCOMO II e pontos de função que são extensivamente baseadas em dados coletados na indústria. Outro problema com a análise de pontos de caso de uso é que a compreensão da equipe sobre o que seja um caso de uso pode produzir estimativas muito díspares. COCOMO II sofre de um problema similar porque é baseada na estimação grosseira do número de linhas de código a serem produzidas ou no uso de tabelas *backfire* (Center for Software Engineering USC, 2000) para converter pontos de função em linhas de código (com diferentes taxas de conversão, dependendo da linguagem usada). Além disso, a análise de pontos de função também depende da capacidade de os estimadores identificar um conjunto correto e completo de requisitos. A conclusão é que cada técnica tem um ponto fraco; entretanto, uma equipe que esteja bem afinada, com registros históricos de estimativas de projetos passados e que aplica os mesmos padrões e convenções quando está desenvolvendo software, poderia produzir estimativas melhores do que uma equipe inexperiente, não importa qual método usem. Todos os métodos dependem de um bom histórico de estimação de esforço e da qualidade dos parâmetros obtidos para o projeto corrente (linhas de código, requisitos funcionais ou casos de uso).

A técnica de análise de pontos de caso de uso foi incorporada pelo Processo Unificado porque tal processo é fortemente baseado em casos de uso. Como dito antes, uma das maiores dificuldades em aplicá-la não é a técnica em si, que é muito simples, mas a falta de padronização sobre o que é efetivamente um caso de uso (Anda, 2001). Mesmo com a ausência de um padrão internacional, a equipe pode criar e usar seu próprio entendimento sobre a granularidade adequada para casos de uso⁸.

A análise de pontos de caso de uso é baseada na quantidade e complexidade dos atores e casos de uso de sistema, o que corresponde ao valor dos *pontos de caso de uso não ajustados* (*unadjusted use case points – UUCP*). A aplicação de fatores técnicos e ambientais leva aos *pontos de caso de uso ajustados* (*use case points – UCP*).

⁷ Antes havia esta explicação: "Disponível em: <www.ifpurg.org>."

⁸ Este livro apresentou algumas sugestões neste sentido nos Capítulos 2 e 3.

O *UUCP* é calculado a partir da estimação da complexidade dos atores, ou *peso dos atores não ajustado* (*unadjusted actor weight – UAW*), e do *peso dos casos de uso não ajustado* (*unadjusted use case weight – UUCW*), como mostrado nas subseções seguintes.

4.2.1 Peso dos atores não ajustado (UAW)

O método para atribuição de complexidade a atores é bastante simples. Cada ator é contado uma única vez, mesmo se ele for associado a vários casos de uso. No caso de atores que são especializações de atores mais genéricos, apenas as especializações mais elementares são contadas, ou seja, atores contados não têm subtipos.

Uma vez que os atores tenham sido identificados, sua complexidade é definida como se segue:

- Atores humanos que interagem com o sistema por meio de uma interface gráfica são considerados complexos e recebem 3 pontos de caso de uso.
- Outros sistemas que interagem com o sistema por meio de um protocolo como TCP/IP e atores humanos que interagem com o sistema por meio de linha de comando são considerados de média complexidade e recebem 2 pontos de caso de uso.
- Outros sistemas que são acessados por interfaces de programação de aplicações (API) são considerados de baixa complexidade e recebem 1 ponto de caso de uso.

O valor do *UAW* é simplesmente a soma dos pontos atribuídos a todos os atores de sistema.

No exemplo da Figura 3.11 os seguintes atores foram considerados:

- Gerente do depósito: 3 pontos.
- Gerente de compras: 3 pontos.
- Gerente de vendas: 3 pontos.
- Comprador: 3 pontos.
- Operadora de cartão de crédito <<system>>: 2 pontos.

Assim, para aquele exemplo, $UAW = 3+3+3+3+2 = 14$.

O uso de *UAW* para estimação de esforço, entretanto, não é aceito unanimemente. Algumas vezes sugere-se que ele pode ser ignorado nas equações. Usualmente, mesmo ferramentas de contagem de *UCP* consideram que *UAW* é uma medida opcional, e usá-la ou não é uma decisão deixada para a equipe.

4.2.2 Peso dos casos de uso não ajustado (UUCW)

Casos de uso também são contados apenas uma vez cada. Apenas processos completos podem ser contados. Isso significa que extensões, variantes e outros fragmentos em geral não devem ser contados.

Na proposta original de Karner (1993), a complexidade de um caso de uso é definida com base no número estimado de *transações* (informação sendo enviada ou recebida pelo sistema), incluindo sequências alternativas (Capítulo 5). Então, casos de uso podem ser caracterizados da seguinte maneira:

- *Casos de uso simples*: não mais do que três transações.
- *Casos de uso médios*: quatro a sete transações.
- *Casos de uso complexos*: mais do que sete transações.

Entretanto, se estivermos trabalhando com casos de uso de alto nível, pode ser difícil saber com certeza quantas transações cada caso de uso realmente tem, e isso faz esta estimativa um tanto arriscada.

Uma forma alternativa de estimar a complexidade de um caso de uso é pelo número de classes necessário para implementar as funções do caso de uso. Assim:

- Casos de uso simples devem ser implementados com cinco classes ou menos.
- Casos de uso médios devem ser implementados com seis a dez classes.
- Casos de uso complexos devem ser implementados com mais de dez classes.

O número de classes necessário para implementar um caso de uso também depende de requisitos detalhados que usualmente não são conhecidos durante a Concepção, considerando-se que os casos de uso ainda não foram expandidos. Isso faz com que essa abordagem também seja um tanto arriscada.

Outra forma de estimar a complexidade de um caso de uso é pela análise de seu risco. Assim:

- *Relatórios* usualmente têm apenas uma ou duas transações e baixo risco, já que eles não alteram os dados. Eles são considerados casos de uso simples.
- *Casos de uso padronizados* como CRUD têm um número de transações que é conhecido e limitado. Seu risco é médio porque, embora sua lógica interna seja conhecida, regras de negócio obscuras podem afetá-los e eles podem ser considerados de média complexidade.
- *Casos de uso não padronizados* têm um número de transações desconhecido e alto risco porque, além do fato de que as regras de negócio podem ser desconhecidas ou mal compreendidas, mesmo a estrutura interna do caso de uso é desconhecida (seu fluxo principal e sequências alternativas). Assim, esse tipo de caso de uso é considerado complexo.

O valor de *UUCW* é calculado pela soma do valor atribuído a cada um dos casos de uso usando-se seguinte critério:

- Casos de uso simples: 5 pontos de caso de uso.
- Casos de uso médios: 10 pontos de caso de uso.
- Casos de uso complexos: 15 pontos de caso de uso.

Aplicando esta técnica ao exemplo da Figura 3.11, existem oito casos de uso estereotipados como relatórios, os quais recebem $8 \times 5 = 40$ pontos de caso de uso. Existem também três casos de uso CRUD (casos de uso padronizados), os quais recebem $3 \times 10 = 30$ pontos. Finalmente, há 10 casos de uso sem nenhum estereótipo, com alto risco, os quais recebem $10 \times 15 = 150$ pontos de caso de uso. Assim, o *UUCW* deste exemplo é $40 + 30 + 150 = 220$.

Existem ainda outras técnicas de contagem, mas a maioria delas é baseada em uma versão detalhada dos casos de uso. Algumas delas são apresentadas na Seção 4.2.9.

4.2.3 Pontos de caso de uso não ajustados (UUCP)

O valor de *UUCP* é calculado como a soma de *UAW* e *UUCW*:

$$UUCP = UAW + UUCW$$

No exemplo corrente, $UUCP = 14 + 220 = 234$.

Como mencionado anteriormente, o peso dos atores poderia ser excluído do cálculo do *UUCP*, e isso faria que *UUCP* fosse imediatamente igual a *UUCW*, ou seja, $UUCP = UUCW = 220$.

4.2.4 Fatores de complexidade técnica (TCF)

A técnica de análise de pontos de caso de uso propõe que o *UUCP* deva ser ajustado por dois critérios: *fatores técnicos* (que são associados ao projeto) e *fatores ambientais* (que são associados à equipe).

Cada fator técnico recebe uma nota de 0 a 5, em que 0 significa nenhuma influência; 3 significa influência nominal; e 5 significa influência máxima no projeto.

Surpreendentemente, esta é a única informação fornecida de maneira geral pela literatura sobre os fatores técnicos de análise de pontos de caso de uso, e isso torna a avaliação desses fatores altamente subjetiva, e mesmo algumas vezes contraditória⁹.

Existem 13 fatores técnicos, cada um dos quais com um peso específico, como mostrado na Tabela 4.3.

Tabela 4.3 Fatores técnicos para ajuste de pontos de função.		
Código	Fator	Peso
T1	Sistema distribuído	2
T2	Tempo de resposta/objetivos de desempenho	1
T3	Eficiência do usuário final	1
T4	Complexidade de processamento interno	1
T5	Design visando reusabilidade de código	1
T6	Facilidade de instalação	0,5
T7	Facilidade de operação	0,5
T8	Portabilidade	2
T9	Design visando facilidade de manutenção	1
T10	Processamento concorrente/paralelo	1
T11	Segurança	1
T12	Acesso de/para código de terceiros	1
T13	Necessidades de treinamento de usuários	1

⁹ Por exemplo, em relação ao fator T12, há autores que dizem que ter mais acesso de código de terceiros implica menos esforço, porque há reuso de código, enquanto outros autores dizem que mais acesso de código de terceiros implica mais esforço, porque será necessário entender o código escrito por outros. Neste livro, uma síntese entre essas duas visões é construída: considera-se bom ter acesso a código bem escrito, não tão bom ter acesso a código que precisa de revisão e pior não ter acesso a qualquer código de terceiros. Reusar código de má qualidade é considerado completamente indesejável (nesses casos usualmente é mais fácil começar a desenvolver do zero).

Assim, cada um dos fatores técnicos tem uma nota entre 0 e 5 que é multiplicada pelo seu peso. A soma desses produtos corresponde ao valor *TFactor*, ou seja, o impacto dos fatores técnicos, que varia de 0 a 70.

O fator de complexidade técnica (*technical complexity fator* – TCF) pode ser calculado ajustando-se o *TFactor* para o intervalo 0,6 a 1,3 por meio da seguinte fórmula:

$$TCF = 0,6 + (0,01 * TFactor)$$

Para uma melhor referência sobre o significado das notas de 0 a 5, sugere-se usar as orientações a seguir, adaptadas da técnica de análise de pontos de função (Albrecht; Gaffney Jr., 1983).

T1 – Sistema distribuído: a arquitetura do sistema é centralizada ou distribuída?

0. A aplicação ignora qualquer aspecto relacionado a processamento distribuído.
1. A aplicação gera dados que serão processados por outros computadores com intervenção humana (por exemplo, planilhas ou arquivos pré-formatados enviados em mídia ou e-mail).
2. Os dados da aplicação são preparados e transferidos automaticamente para processamento em outros computadores.
3. O processamento da aplicação é distribuído e os dados são transferidos apenas em uma direção.
4. O processamento da aplicação é distribuído e os dados são transferidos em ambas as direções.
5. Os processos da aplicação devem ser executados nos núcleos de processamento ou computadores mais apropriados, o que é determinado dinamicamente.

T2 – Tempo de resposta/objetivos de desempenho: qual é a importância do tempo de resposta da aplicação para seus usuários?

0. Nenhum requisito especial de desempenho foi definido pelo cliente.
1. Requisitos de desempenho foram estabelecidos e revisados, mas nenhuma ação especial precisa ser tomada.
2. Tempo de resposta e taxas de transferência são críticos durante as horas de pico. Nenhum design especial para núcleos de processamento é necessário. O prazo para a maioria dos processamentos é o dia seguinte.
3. Tempo de resposta e taxas de transferência são críticos durante as horas de pico. Nenhum design especial para núcleos de processamento é necessário. Requisitos referentes a prazos para comunicação com sistemas interfaceados são restritivos.
4. Em adição ao item 3, requisitos de desempenho são suficientemente restritivos para exigir tarefas de análise de desempenho durante o design.
5. Em adição a 4, ferramentas de análise devem ser usadas durante o design, desenvolvimento e/ou implementação para garantir que os requisitos de desempenho sejam obtidos.

T3 – Eficiência do usuário final: a aplicação é projetada de forma que os usuários finais apenas façam seu trabalho ou ela é projetada para que eles melhorem sua eficiência?

0. A aplicação não necessita de nenhum dos itens abaixo.
1. A aplicação necessita de um a três dos itens abaixo.
2. A aplicação necessita de quatro a cinco dos itens abaixo.
3. A aplicação necessita de seis ou mais dos itens abaixo, mas não há requisitos relacionados à eficiência do usuário.
4. A aplicação necessita de seis ou mais dos itens abaixo, e os requisitos de eficiência do usuário são tão fortes que o design deve incluir características para minimizar a digitação, maximizar defaults, usar templates etc.
5. A aplicação necessita de seis ou mais dos itens abaixo, e os requisitos de eficiência do usuário são tão fortes que atividades de projeto devem incluir ferramentas e processos especiais para demonstrar que os objetivos de eficiência foram obtidos.

Os seguintes itens devem ser considerados para a avaliação do item de eficiência do usuário final¹⁰:

- Ajuda navegacional (por exemplo, menus gerados dinamicamente, hipermídia adaptativa etc.).
- Ajuda e documentação *on-line*.
- Movimentação de cursor automatizada.
- Teclas de função predefinidas.
- Tarefas em lote submetidas a partir de transações *on-line*.
- Alto uso de cores e destaque visual nas telas.
- Minimização do número de telas para obter as metas de negócio.
- Suporte bilíngue (conta como quatro itens).
- Suporte multilíngue (conta como seis itens).

T4 – Complexidade de processamento interno: a aplicação necessita de algoritmos complexos?

0. Nenhuma das opções a seguir.
1. Uma das opções a seguir.
2. Duas das opções a seguir.
3. Três das opções a seguir.
4. Quatro das opções a seguir.
5. Todas as cinco opções a seguir.

As seguintes opções devem ser consideradas para avaliar a necessidade de processamento interno complexo:

- Controle crítico (por exemplo, processamento de auditoria especial) e/ou processamento de segurança específico para a aplicação.

¹⁰ Alguns itens, como “menus” e “scrolling”, que originalmente eram considerados pela análise de pontos de função, foram removidos da lista porque são considerados muito triviais para as aplicações atuais.

- Processamento lógico extensivo.
- Processamento matemático extensivo.
- Muito processamento de exceção resultante de transações incompletas que necessitam ser reprocessadas, como transações incompletas de máquinas de saque automático causadas por interrupção de comunicação, valores de dados perdidos ou falha na alteração de dados.
- Processamento complexo para gerenciar possibilidades de entrada e saída múltiplas, como multimídia ou independência de dispositivo.

T5 – Design visando reusabilidade de código: a aplicação é projetada de forma que seu código e artefatos sejam altamente reusáveis?

0. Não há nenhuma preocupação a respeito de produzir código reusável.
1. Código reusável é gerado para uso dentro do mesmo projeto.
2. Menos de 10% da aplicação deve considerar mais do que as necessidades do usuário.
3. 10% ou mais da aplicação deve considerar mais do que as necessidades do usuário.
4. A aplicação deve ser especificamente empacotada e/ou documentada para facilitar o reuso e ser personalizável pelo usuário em nível de código fonte.
5. A aplicação deve ser especificamente empacotada e/ou documentada para facilitar o reuso e ser personalizável pelo usuário com o uso de parâmetros.

T6 – Facilidade de instalação: a aplicação será projetada de forma que a instalação seja automática (por exemplo, no caso de usuários com capacidade técnica baixa ou desconhecida), ou não há preocupações especiais a respeito disso?

0. O cliente não estabeleceu uma consideração especial e um *setup* especial é necessário para a instalação.
1. O cliente não estabeleceu considerações especiais, mas um *setup* especial é necessário para a instalação.
2. O cliente estabeleceu requisitos para conversão de dados e instalação, e guias de conversão e instalação devem ser fornecidos e testados. O impacto da conversão de dados no projeto não é considerado importante.
3. O cliente estabeleceu requisitos para conversão de dados e instalação, e guias de conversão e instalação devem ser fornecidos e testados. O impacto da conversão de dados no projeto é considerável.
4. Em adição a 2, ferramentas para conversão automática e instalação devem ser fornecidas e testadas.
5. Em adição a 3, ferramentas para conversão automática e instalação devem ser fornecidas e testadas.

T7 – Facilidade de operação¹¹: Existem requisitos especiais a respeito da operação do sistema?

0. Nenhuma consideração especial sobre a operação do sistema, além dos procedimentos de *backup* usuais, foi estabelecida pelo usuário.
1. Um dos itens a seguir se aplica ao sistema.
2. Dois dos itens a seguir se aplicam ao sistema.
3. Três dos itens a seguir se aplicam ao sistema.
4. Quatro dos itens a seguir se aplicam ao sistema.
5. A aplicação é projetada para operar de modo não supervisionado. “Não supervisionado” significa que a intervenção humana não é necessária para manter o sistema operacional, mesmo se panes ocorrerem, exceto talvez para o *startup* inicial e o desligamento final.

Para a avaliação da facilidade de operação, os seguintes itens devem ser considerados:

- Processos efetivos para inicialização, *backup* e recuperação devem ser fornecidos, mas a intervenção do operador é ainda necessária.
- Processos efetivos para inicialização, *backup* e recuperação devem ser fornecidos, e nenhuma intervenção do operador é necessária (conta como dois itens).
- A aplicação deve minimizar a necessidade de armazenamento de dados em meios *off-line* (por exemplo, fitas).
- A aplicação deve minimizar a necessidade de lidar com papel.

T8 – Portabilidade: A aplicação ou suas partes são projetadas para funcionar em mais do que uma plataforma?

0. Não há requisito de usuário que considere a necessidade de instalar a aplicação em mais do que uma plataforma.
1. O design deve considerar a necessidade de o sistema operar em diferentes plataformas, mas a aplicação deve ser projetada para operar apenas em ambientes de hardware e software *idênticos*.
2. O design deve considerar a necessidade de o sistema operar em diferentes plataformas, mas a aplicação deve ser projetada para operar apenas em ambientes de hardware e software *similares*.
3. O design deve considerar a necessidade de o sistema operar em diferentes plataformas, e a aplicação deve ser projetada para operar em ambientes de hardware e software *heterogêneos*.
4. Em adição a 1 ou 2, a documentação e o plano de manutenção devem ser elaborados e testados para suportar a operação em múltiplas plataformas.

¹¹ Ocasionalmente, este item é referenciado como “usabilidade”, mas o conceito original de usabilidade já foi considerado no fator T3, “eficiência do usuário final”. Preferimos interpretar este fator T7 como “facilidade de operação” por sua compatibilidade com os fatores técnicos de análise de pontos de função, e também para evitar confusão e redundância com o fator de eficiência do usuário final.

5. Em adição a 3, a documentação e o plano de manutenção devem ser elaborados e testados para suportar a operação em múltiplas plataformas.

T9 – Design visando facilidade de manutenção: o cliente requer que a aplicação seja fácil de manter e mudar no futuro?

0. Nenhum dos itens a seguir.
1. Um dos itens a seguir.
2. Dois dos itens a seguir.
3. Três dos itens a seguir.
4. Quatro dos itens a seguir.
5. Cinco ou mais dos itens a seguir.

Para avaliar este fator, os seguintes itens devem ser considerados:

- Estrutura de relatório flexível deve ser fornecida para lidar com consultas simples, como operadores lógicos binários aplicados a apenas um arquivo lógico (conta como um item).
- Estrutura de relatório flexível deve ser fornecida para lidar com consultas de complexidade média tais como operadores lógicos binários aplicados a mais do que um arquivo lógico (conta como dois itens).
- Estrutura de relatório flexível deve ser fornecida para lidar com consultas de alta complexidade, como combinações de operadores lógicos binários aplicados a um ou mais arquivos lógicos (conta como três itens).
- Dados de controle de negócio são mantidos em tabelas gerenciadas pelo usuário com acesso interativo *on-line*, mas mudanças somente são efetivadas no dia seguinte (conta como um item).
- Dados de controle de negócio são mantidos em tabelas gerenciadas pelo usuário com acesso interativo *on-line*, e mudanças são efetivadas imediatamente (conta como dois itens).

T10 – Processamento concorrente/paralelo: a aplicação deve ser projetada para lidar com problemas relacionados à concorrência, como, por exemplo, compartilhamento de dados e recursos?

0. Nenhum acesso concorrente aos dados é esperado.
1. Acesso concorrente aos dados é esperado algumas vezes.
2. Acesso concorrente aos dados é esperado frequentemente.
3. Acesso concorrente aos dados é esperado o tempo todo.
4. Em adição a 3, o usuário indica que muitos acessos múltiplos vão ocorrer, forçando tarefas de análise de desempenho e resolução de *deadlock* durante o design.
5. Em adição a 4, o design requer o uso de ferramentas especiais para controlar o acesso.

T11 – Segurança: as necessidades de segurança são apenas nominais ou há requisição de especificações adicionais e design especial?

0. Não há requisitos especiais em relação à segurança.
1. A necessidade de segurança deve ser levada em conta no design.
2. Em adição a 1, a aplicação deve ser projetada de forma que só possa ser acessada por usuários autorizados.
3. Em adição a 2, o acesso ao sistema deve ser controlado e auditado.
4. Em adição a 3, um plano de segurança deve ser elaborado e testado para suportar o controle de acesso à aplicação.
5. Em adição a 4, um plano de segurança deve ser elaborado e testado para suportar auditoria.

T12 – Acesso de/para código de terceiros: a aplicação vai usar código já desenvolvido, tal como componentes *Commercial Off-The-Shelf* (COTS), *frameworks* ou bibliotecas? Alto reuso de software de boa qualidade reduz o valor desse item, já que implica menos esforço de desenvolvimento.

0. Código preexistente altamente confiável será usado extensivamente para o desenvolvimento da aplicação.
1. Código preexistente altamente reusável será usado em pequenas partes da aplicação.
2. Código preexistente que eventualmente precisa ser ajustado vai ser usado extensivamente para o desenvolvimento da aplicação.
3. Código preexistente que eventualmente precisa ser ajustado será usado em pequenas partes da aplicação.
4. Código preexistente que necessita ser consertado ou que é difícil de entender será usado na aplicação.
5. Nenhum código preexistente ou código de qualidade questionável será usado na aplicação.

T13 – Necessidades de treinamento de usuários: a aplicação será fácil de usar ou treinamento extensivo deve ser dado aos futuros usuários.

0. Não há requisitos específicos a respeito de treinamento de usuários.
1. Requisitos específicos de treinamento de usuários foram mencionados.
2. Existem requisitos formais específicos para treinamento de usuários e a aplicação deve ser projetada para facilitar o treinamento.
3. Existem requisitos formais específicos para treinamento de usuários e a aplicação deve ser projetada para suportar usuários com diferentes níveis de treinamento.
4. Um plano de treinamento detalhado deve ser elaborado para a fase de Transição e executado.
5. Em adição a 4, usuários estão geograficamente distribuídos.

Aplicando os fatores técnicos ao exemplo corrente, um resultado similar ao apresentado na Tabela 4.4 poderia ser obtido.

Tabela 4.4 Avaliação dos fatores técnicos de Livir.					
Código	Fator	Peso	Nota	Peso × Nota	Justificativa
T1	Sistema distribuído	2	0	0	A aplicação ignora quaisquer aspectos relacionados a processamento distribuído.
T2	Tempo de resposta/objetivos de desempenho	1	2	2	Tempo de resposta e taxas de transferência são críticos durante horas de pico. Nenhum design especial de núcleos de processador é necessário. O prazo para a maioria dos processos é o dia seguinte.
T3	Eficiência do usuário final	1	3	3	A aplicação necessita dos seguintes itens, mas não há requisitos especiais relacionados com eficiência do usuário: • Ajuda navegacional (por exemplo, menus dinamicamente gerados, hipermídia adaptativa etc.) • Ajuda e documentação <i>on-line</i>. • Alto uso de cores e destaque visual em telas. • Minimização do número de telas para atingir objetivos de negócio. • Suporte multilíngual (conta como seis itens).
T4	Complexidade de processamento interno	1	0	0	Nenhuma das opções se aplica.
T5	Design visando reusabilidade de código	1	0	0	Não há preocupação em produzir código reusável.
T6	Facilidade de instalação	0,5	0	0	O cliente não estabeleceu considerações especiais e nenhum <i>setup</i> especial é necessário para a instalação.
T7	Facilidade de operação	0,5	5	2,5	A aplicação é projetada para operar de forma não supervisionada. “Não supervisionada” significa que a intervenção humana não é necessária para manter o sistema operacional, mesmo se ocorrerem panes, exceto, talvez, para o primeiro <i>startup</i> ou desligamento final. Uma das características da aplicação é recuperação automática de erros.
T8	Portabilidade	2	3	6	O design deve considerar a necessidade de o sistema operar em diferentes plataformas, mas a aplicação cliente deve ser projetada para operar em ambientes de hardware e software heterogêneos.
T9	Design visando facilidade de manutenção	1	0	0	Nenhum dos itens se aplica.
T10	Processamento concorrente/paralelo	1	1	1	Acesso concorrente a dados é esperado eventualmente.
T11	Segurança	1	3	3	A necessidade de segurança deve ser levada em conta durante o design. A aplicação deve ser projetada de forma que possa ser acessada apenas por usuários autorizados. O acesso ao sistema será controlado e auditado.
T12	Acesso de/para código de terceiros	1	5	5	Nenhum código preexistente será usado na aplicação.
T13	Necessidades de treinamento de usuários	1	0	0	Não há requisitos específicos para treinamento de usuários.
TFactor				22,5	

O valor dos fatores técnicos, TCF , para o exemplo corrente é obtido por:

$$TCF = 0,6 + (0,01 * 22,5) = 0,825$$

Como este valor é menor do que 1, os fatores técnicos desse projeto indicam um tempo de desenvolvimento que é menor do que o nominal (82,5% do nominal). Entretanto, os fatores ambientais ainda devem ser determinados.

4.2.5 Fatores ambientais (EF)

Um aspecto que distingue a análise de pontos de caso de uso da análise de pontos de função é que a análise de pontos de caso de uso considera um fator de ajuste para as características da equipe de desenvolvimento. Assim, o mesmo projeto com os mesmos fatores técnicos pode ter um número diferente de pontos de caso de uso ajustados para diferentes equipes.

Existem oito fatores ambientais que avaliam o ambiente de trabalho. Cada um tem seu próprio peso e dois deles têm pesos negativos. Novamente, as notas variam de 0 a 5, mas seu significado é um pouco diferente do das notas atribuídas aos fatores técnicos. Enquanto as notas dos fatores técnicos avaliam a *influência* desses fatores no projeto, as notas dos fatores ambientais avaliam a *qualidade* desses fatores no ambiente de desenvolvimento. Por exemplo, uma nota 0 para o fator “motivação” indica que a equipe não está motivada, uma nota 3 indica motivação média e uma nota 5, que a equipe está altamente motivada.

Os fatores ambientais são definidos na Tabela 4.5.

Tabela 4.5 Fatores ambientais para ajuste de pontos de caso de uso.		
Código	Fator	Peso
E1	Familiaridade com o processo de desenvolvimento	1,5
E2	Experiência na aplicação	0,5
E3	Experiência com orientação a objetos	1
E4	Experiência do analista líder	0,5
E5	Motivação	1
E6	Estabilidade de requisitos	2
E7	Trabalhadores em tempo parcial	-1
E8	Dificuldade com a linguagem de programação	-1

Assim, as notas multiplicadas pelo peso dos fatores ambientais podem variar de -10 a 32,5. Este valor é chamado de $EFactor$.

$EFactor$ é ajustado para o intervalo 0,425 a 1,7, gerando o fator ambiental (*environmental factor* – EF) pela seguinte fórmula:

$$EF = 1,4 - (0,03 * EFactor)$$

Ribu (2001) apresenta uma referência para atribuição de notas aos fatores ambientais, que é descrita adiante com algumas adaptações.

E1 – familiaridade com o processo de desenvolvimento: este fator avalia a experiência da equipe com o processo de desenvolvimento sendo utilizado. Originalmente, o fator mencionava o Processo Unificado, e alguns autores inclusive mencionavam UML aqui. Mas isso foi ajustado para refletir o fato de que outros processos e linguagens de modelagem podem ser usados também:

0. A equipe não tem experiência com o processo de desenvolvimento ou não usa qualquer processo.
1. A equipe tem conhecimento teórico sobre o processo de desenvolvimento, mas não experiência.
2. Alguns membros da equipe já tiveram experiência com o processo em um projeto.
3. Alguns membros da equipe já usaram o processo em mais de um projeto.
4. Até metade da equipe já usou o processo em vários projetos.
5. Mais da metade da equipe já usou o processo em vários projetos.

E2 – Experiência na aplicação: este fator avalia a familiaridade da equipe com a área ou domínio de aplicação. Por exemplo, se a aplicação é sobre comércio eletrônico, alguém na equipe já trabalhou com sistemas na mesma área?

0. Nenhum membro da equipe tem qualquer experiência em projetos na mesma área.
1. Alguns membros da equipe têm de 6 a 12 meses de experiência em projetos na mesma área.
2. Alguns membros da equipe têm de 12 a 18 meses de experiência em projetos na mesma área.
3. A maioria dos membros da equipe tem de 18 a 24 meses de experiência em projetos na mesma área.
4. A maioria dos membros da equipe tem mais de dois anos de experiência em projetos na mesma área.
5. Todos os membros da equipe tem mais do que dois anos de experiência em projetos na mesma área.

E3 – Experiência em orientação a objetos: este fator não deve ser confundido com *familiaridade com o processo de desenvolvimento* (E1) nem com *experiência na aplicação* (E2): ele trata da experiência da equipe em realizar análise, modelagem, design e programação orientados a objetos independentemente da área de aplicação ou do processo utilizado:

0. A equipe não tem experiência com técnicas orientadas a objetos.
1. Todos os membros da equipe têm alguma experiência com técnicas orientadas a objetos (até um ano).
2. A maioria dos membros da equipe tem de 12 a 18 meses de experiência com técnicas orientadas a objetos.
3. Todos os membros da equipe têm experiência entre 12 e 18 meses ou a maioria dos membros tem experiência entre 18 e 24 meses em técnicas orientadas a objetos.

4. A maioria dos membros da equipe tem mais do que dois anos de experiência em técnicas orientadas a objetos.
5. Todos os membros da equipe têm mais do que dois anos de experiência em técnicas orientadas a objetos.

E4 – Experiência do analista líder: este fator mede a experiência do analista líder com análise de requisitos e modelagem orientada a objetos:

0. O analista líder não tem experiência.
1. O analista líder tem experiência prévia em um único projeto semelhante.
2. O analista líder tem cerca de um ano de experiência em mais do que um projeto similar.
3. O analista líder tem cerca de dois anos de experiência em projetos similares.
4. O analista líder tem mais do que dois anos em projetos similares.
5. O analista líder tem mais do que três anos de experiência em projetos variados.

E5 – Motivação: este fator descreve a motivação da equipe¹²:

0. A equipe não tem motivação alguma. Sem supervisão constante, a equipe se torna improdutiva. A equipe faz apenas o que é estritamente solicitado.
1. A equipe tem muito pouca motivação. Supervisão constante é necessária para manter a produtividade em níveis aceitáveis.
2. A equipe tem pouca motivação. Intervenções da gerência são necessárias de tempos em tempos para manter a produtividade.
3. A equipe tem alguma motivação. Usualmente, a equipe tem iniciativa, mas a intervenção da gerência é ainda necessária esporadicamente para manter a produtividade.
4. A equipe é bem motivada. A equipe usualmente se autogerencia, mas a existência de supervisão é ainda necessária porque a produtividade pode ser perdida sem ela.
5. A equipe é altamente motivada. Mesmo sem supervisão, cada um sabe o que deve ser feito, e o ritmo é mantido indefinidamente.

E6 – Requisitos estáveis: este fator avalia se a equipe foi capaz de manter os requisitos estáveis em projetos passados, minimizando sua mudança durante o projeto. Este fator pode variar de projeto para projeto porque há domínios nos quais requisitos mudam independentemente da capacidade dos analistas. Este fator deve avaliar apenas as mudanças no software que foram causadas pela inabilidade da equipe em descobrir os requisitos corretos:

0. Não há histórico sobre estabilidade de requisitos ou no passado análise malfeita causou grandes mudanças em requisitos após o projeto iniciar.

¹² Certamente, este é um dos fatores ambientais mais difíceis de avaliar, porque a real motivação da equipe pode estar mascarada. A sugestão de Ribu (2001) parece ajudar muito pouco em relação a estabelecer uma nota para este fator, e nós tomamos a liberdade de reinterpretar a referência.

1. Requisitos foram predominantemente instáveis no passado. Clientes solicitaram muitas mudanças causadas principalmente por requisitos incompletos ou incorretos.
2. Requisitos foram instáveis no passado. Clientes pediram algumas mudanças causadas por requisitos incompletos ou incorretos.
3. Requisitos foram relativamente estáveis no passado. Clientes pediram mudanças em funcionalidades secundárias com alguma regularidade. Mudanças em funcionalidades principais foram solicitadas raramente.
4. Requisitos foram predominantemente estáveis no passado. Clientes pediram pequenas mudanças, especialmente estéticas. Mudanças em funcionalidades principais ou secundárias foram raras.
5. Requisitos foram totalmente estáveis no passado. Pequenas mudanças eventuais não tiveram impacto no projeto.

E7 – Trabalhadores em tempo parcial: Este é um fator negativo, ou seja, ao contrário dos seis primeiros fatores ambientais, valores altos aqui representam mais tempo de desenvolvimentos, e não menos. Uma equipe com muitos membros envolvidos em outros projetos ou atividades será menos produtiva do que uma equipe dedicada:

0. Nenhum membro da equipe trabalha em tempo parcial.
1. Até 10% da equipe trabalha em tempo parcial.
2. Até 20% da equipe trabalha em tempo parcial.
3. Até 40% da equipe trabalha em tempo parcial.
4. Até 60% da equipe trabalha em tempo parcial.
5. Mais de 60% da equipe trabalha em tempo parcial.

E8 – Dificuldade com a linguagem de programação: este é outro fator negativo, o que significa que valores altos aqui são ruins para o tempo de desenvolvimento.

0. Todos os membros da equipe são programadores muito experientes.
1. A maioria dos membros da equipe tem pelo menos dois anos de experiência em programação.
2. Todos os membros da equipe têm pelo menos 18 meses de experiência em programação.
3. A maioria dos membros da equipe tem pelo menos 18 meses de experiência em programação.
4. Alguns membros da equipe têm alguma experiência em programação (não mais de um ano).
5. Todos os membros da equipe são programadores inexperientes.

Para continuar com o exemplo Livir, vamos assumir que a empresa é formada por um grupo de cinco profissionais que recentemente obtiveram seu diploma em Sistemas

de Informação e que estudaram técnicas de orientação a objetos, mas não têm experiência profissional significativa. Três deles estão totalmente dedicados ao projeto, enquanto os outros dois têm outras atividades. Livir será o primeiro projeto que eles farão juntos profissionalmente. Com esta empresa fictícia, a avaliação dos fatores ambientais seria algo semelhante à Tabela 4.6.

Tabela 4.6 Fatores ambientais para uma empresa fictícia desenvolvendo o projeto Livir.

Código	Fator	Peso	Nota	Peso × Nota	Justificativa
E1	Familiaridade com o processo de desenvolvimento	1,5	1	1,5	A equipe tem conhecimento teórico sobre o processo de desenvolvimento, mas não tem prática profissional.
E2	Experiência na aplicação	0,5	0	0	Nenhum dos membros da equipe tem experiência prévia em projetos similares.
E3	Experiência com orientação a objetos	1	3	3	Todos os membros da equipe têm entre 12 e 18 meses de experiência (acadêmica) com técnicas orientadas a objetos.
E4	Experiência do analista líder	0,5	0	0	O analista líder não tem experiência.
E5	Motivação	1	5	5	A equipe está altamente motivada. Mesmo sem supervisão, cada membro da equipe sabe seu trabalho e mantém o ritmo indefinidamente.
E6	Estabilidade de requisitos	2	0	0	Não há nenhum histórico confiável para a equipe.
E7	Trabalhadores em tempo parcial	-1	3	-3	40% da equipe trabalha em tempo parcial.
E8	Dificuldade com a linguagem de programação	-1	5	-5	Embora eles tenham experiência acadêmica, todos os programadores são novatos.
EFactor				1,5	

Aplicando-se fórmula de ajuste aos fatores ambientais da empresa, o seguinte resultado é obtido:

$$EF = 1,4 - (0,03 * 1,5) = 1,355$$

Isso indica que os fatores ambientais são ruins para esta equipe de iniciantes e que seu esforço será cerca de 35% maior do que o nominal.

4.2.6 Pontos de caso de uso ajustados (UCP)

Uma vez que os pesos dos atores e casos de uso, bem como os fatores ambientais e técnicos, tenham sido calculados, os pontos de caso de uso ajustados podem ser obtidos aplicando-se uma simples multiplicação:

$$UCP = UUCP * TCF * EF$$

Para o exemplo corrente, obtém-se:

$$UCP = 234 * 0,825 * 1,355 = 261,583$$

4.2.7 Esforço

O objetivo-chave de todas as técnicas de estimação é prever o *esforço* necessário para desenvolver um projeto. Em primeiro lugar, é necessário definir a *extensão* do esforço que será estimado. O método COCOMO II, por exemplo, quando aplicado com o Processo Unificado, estima o esforço que será gasto apenas durante as fases de Elaboração e Construção.

Mas, além disso, o esforço inclui atividades como as administrativas ou de vendas? Ou ele inclui apenas o esforço despendido nas atividades de desenvolvimento? A literatura de pontos de caso de uso usualmente não é tão detalhada quando a de COCOMO II e a análise de pontos de função a respeito do tipo de esforço que está sendo estimado. Como a análise de pontos de caso de uso é baseada na análise de pontos de função MK II (UKSMA Metrics Practices Committee, 1998), podemos assumir que, por default, as mesmas definições se aplicam, como “O esforço do projeto deve incluir a equipe que esteja claramente alocada ao projeto e que trabalha de acordo com os requisitos do projeto. O tempo de um usuário sendo entrevistado, por exemplo, não deveria ser incluído”¹³. Similarmente, a seguinte definição também poderia ser aplicada: “O esforço de trabalho deveria ser medido em unidades de horas de trabalho ‘produtivas’ ou ‘líquidas’, o que poderia ser definido como uma hora de trabalho por uma pessoa, incluindo as paradas normais, mas excluindo os grandes intervalos, como, por exemplo, o do almoço”. MK II também estabelece que um projeto começa quando “foi acordado e aprovado que esforço será despendido para entregar uma solução de SI¹⁴ [...] um ou mais funcionários tenham sido alocados ao projeto, e iniciaram o trabalho nele” (isto acontece durante a fase de Concepção), e que um projeto termina quando “a aplicação é ativada ao ser colocada no ambiente de produção pela primeira vez e os usuários começam a usá-la” (isso acontece durante a fase de Transição).

O esforço que é calculado usualmente deve incluir todos os riscos e situações inesperadas. Estimação, entretanto, não é uma ciência precisa. Em algumas situações, as coisas podem sair do controle e ajustes podem ser necessários em tempo real para manter o projeto nos trilhos. Pode-se dizer que o cronograma final poderia definir o *tempo máximo tolerável* para a entrega de um produto com o *mínimo escopo aceitável*. Isto é, se tivermos sorte e os riscos causarem poucos problemas, podemos entregar o produto mais cedo ou entregar um produto melhor do que o mínimo esperado.

Uma vez que o número de pontos de caso de uso tenha sido calculado, ele pode ser transformado em uma medida de esforço pelo uso do índice de produtividade da *equipe* (*Team Productivity Index* – TPI). Se o TPI for medido como o tempo que um membro da equipe gasta com um ponto de caso de uso¹⁵, então o esforço pode ser dado por:

$$E = UCP * TPI$$

¹³ Tradução do autor.

¹⁴ Sistema de Informação.

¹⁵ Opcionalmente, o TPI poderia ser expresso como o número de pontos de caso de uso que um membro da equipe consegue produzir em uma dada unidade de tempo. Neste caso, a fórmula deveria ser $E = UCP / TPI$.

Como *UCP* é uma medida independente de escala, deve-se decidir em qual escala o esforço será medido. A literatura de pontos de caso de uso normalmente utiliza *funcionário-hora* por ponto de caso de uso. Entretanto, outros métodos, como COCOMO II (Boehm, 2000), preferem *funcionário-mês*.

Já que um grande número de fórmulas usualmente aplicadas à produtividade na indústria de software é definido em termos de *funcionário-mês* e como a escolha da unidade pode afetar os resultados, este livro adota funcionários-mês como a medida de produtividade, mas refere-se a funcionários-hora quando necessário.

Considerando que um mês corresponde a 158 horas de trabalho em média (fins de semana e feriados descontados), então x funcionários-hora por ponto de caso de uso correspondem a $x/158$ funcionários-mês por ponto de caso de uso.

Originalmente, Karner (1993) estimou que 20 funcionários-hora (aproximadamente 0,127 funcionário-mês por ponto de caso de uso)¹⁶ seriam uma boa estimativa quando não houvesse um histórico confiável. Mais tarde, Ribu (2001) determinou que este valor poderia variar de 15 a 30 funcionários-hora (0,095 a 0,190 funcionários-mês) por ponto de caso de uso.

O valor do *TPI* também pode ser estimado tomando-se projetos passados, olhando-se para o tempo total de pessoal gasto e o dividindo-se pelo número de pontos de caso de uso. Assim, por exemplo, se o esforço total gasto em um dado projeto foi de 9 funcionários-mês e o *UCP* total do projeto foi 60, então o índice de produtividade da equipe neste caso foi de 9/60, ou seja, 0,15 funcionário-mês por ponto de caso de uso.

Outro trabalho (Schneider; Winters, 1998) propõe que o número de fatores ambientais E1 a E6 com nota menor do que 3 deve ser adicionado ao número de fatores ambientais de E7 e E8 com nota acima de 3 e:

- Se o resultado for 2 ou menos, assumir 20 funcionários-hora (0,127 funcionário-mês) por ponto de caso de uso.
- Se o resultado for 3 ou 4, assumir 28 funcionários-hora (0,177 funcionário-mês) por ponto de caso de uso.
- Se o resultado for maior do que 4, os fatores ambientais apresentam um risco muito alto para o projeto e devem ser melhorados antes que qualquer compromisso seja assumido, ou então assumir 36 funcionários-hora (0,228 funcionário-mês) por ponto de caso de uso.

No exemplo corrente, existem cinco fatores ambientais ruins: E1, E2, E4, E6 e E8. A recomendação, neste caso, seria tentar melhorar a qualidade do ambiente antes de desenvolver qualquer projeto. Entretanto, se a equipe decidir assumir o risco, o projeto deveria assumir a pior estimativa de 0,228 funcionário-mês por ponto de caso de uso. Assim, o esforço total do projeto Livir seria $E = 261,583 * 0,228 \approx 59,64$ funcionários-mês.

¹⁶ Lembre-se de que um caso de uso complexo tem 15 pontos de caso de uso e assim requer cerca de 300 funcionários-hora para ser totalmente desenvolvido, de acordo com Karner (1993).

4.2.8 Tempo linear e tamanho médio da equipe

O valor do esforço obtido para o exemplo na última seção (59,64 funcionários-mês) refere-se ao esforço total para desenvolver o projeto desde seu início até a entrega do produto. Entretanto, mais do que um membro da equipe estará trabalhando no projeto e assim o tempo linear (tempo de calendário com fins de semana e feriados descontados) seria provavelmente bem menor.

Existe uma fórmula simples que permite aproximar o tempo linear ideal de um projeto a partir de seu esforço total (McConnel, 1996). A fórmula é a seguinte:

$$T = 2,5 * \sqrt[3]{E}$$

Nesta fórmula, T é o tempo linear ideal para desenvolver o projeto.

Esta fórmula só funciona se E for medido em termos de *funcionário-mês*. O uso de outras unidades como funcionário-semana ou funcionário-hora causaria distorções no resultado em razão do uso da raiz cúbica.

Outras fórmulas baseadas em pontos de função (Jones, 2007) ou linhas de código (Boehm, 2000) estão disponíveis. Mas para ser aplicadas com pontos de caso de uso, algumas conversões seriam necessárias. Se uma aproximação mais acurada for desejada, talvez isso valha o esforço, porque as fórmulas de Boehm e Jones usam informação sobre a complexidade do projeto e sobre a capacidade da equipe como parâmetros para produzir o tempo linear ideal estimado.

Aplicando-se a fórmula simples ao exemplo corrente obtém-se:

$$T = 2,5 * \sqrt[3]{59,64} \approx 9,768 \text{ meses}$$

O tamanho médio da equipe (P) é calculado por:

$$P = E / T$$

Para o exemplo corrente, obtém-se:

$$P = 59,64 / 9,768 \approx 6,106$$

Assim, o tamanho médio da equipe, neste caso, deve ser de aproximadamente seis pessoas. Em resumo, o projeto poderia ser desenvolvido idealmente por seis pessoas durante 10 meses. Para este exemplo, a empresa poderia decidir contratar um novo profissional (especialmente se fosse um mais experiente que pudesse ajudar a melhorar as notas dos fatores ambientais); tentar desenvolver o projeto com cinco pessoas levaria mais tempo do que o ideal. Para este exemplo, seria possível calcular o tempo que uma equipe de cinco pessoas levaria para desenvolver o projeto como $59,64/5 = 11,928$. Neste caso, o projeto poderia ser completado por cinco pessoas em aproximadamente 12 meses. Diz-se “seria possível”, porque o número de pessoas considerado é menor do que o tamanho médio ideal para a equipe. Entretanto, não seria realista aplicar esta divisão para calcular o tempo linear para uma equipe maior do que a ideal. Por exemplo, não seria realista assumir que 20 pessoas poderiam completar o projeto em $59,64/20 = 2,982$ meses. Qualquer tamanho de equipe maior do que o ideal implica que o esforço (E) seja ajustado para um valor também mais alto. O multiplicador de esforço SCED (cronograma de desenvolvimento requerido)

de COCOMO II indica que realizar um projeto em 85% do tempo ideal implica adicionar 14% ao esforço total e realizar um projeto em 75% do tempo ideal implica adicionar 43% ao esforço total (Boehm, 2000).

Pode parecer que esses tempos são muito altos para o projeto apresentado no exemplo. Entretanto, deve-se considerar que os fatores ambientais são muito ruins, já que a equipe tem muito pouca experiência. Eles vão cometer erros e possivelmente terão de refazer muito trabalho. Mais importante, a falta de experiência pode levar a equipe a entender mal os requisitos e ter de consertá-los depois, aumentando o tempo de desenvolvimento significativamente.

Se, em vez desses novatos, tivéssemos uma *equipe dos sonhos* com as melhores notas possíveis nos fatores ambientais, então EF poderia ser 0,425 (o melhor valor possível). Nesse caso, $UCP = 234 * 0,825 * 0,425 = 82,046$. Usando a recomendação de Schneider e Winters (1998), pode-se assumir que esta equipe dos sonhos poderia trabalhar a uma taxa de 20 funcionários-hora ou 0,127 funcionário-mês por ponto de função. Assim, $E = 82,046 * 0,127 = 10,42$. Aplicando-se a fórmula para o tempo linear e o tamanho médio da equipe, chega-se em $T = 5,46$ e $P = 1,908$. Isto é, uma equipe dos sonhos com duas pessoas poderia completar o projeto em cerca de cinco meses e meio. Isso parece uma estimativa bem razoável considerando-se o tamanho do projeto e a capacidade da equipe.

Entretanto, é importante mencionar que essas fórmulas são um tanto “mágicas”, porque existe uma grande variedade de projetos e equipes de desenvolvimento. Antes de usá-las seriamente em uma empresa, é sempre útil testar e ajustar as fórmulas para a realidade local.

4.2.9 Métodos de contagem para casos de uso detalhados

A técnica de análise de pontos de caso de uso apresentada nas seções anteriores considera apenas uma estimativa da complexidade dos casos de uso porque ela trabalha com casos de uso de alto nível, os quais são representados apenas pelo seu nome e, eventualmente, uma ou duas frases explicativas.

Existem técnicas para contagem de pontos de caso de uso para casos de uso expandidos (explicados no Capítulo 5), mas sua utilidade é mais limitada porque, com o Processo Unificado, casos de uso usualmente somente são detalhados durante as iterações. Durante a Concepção, a equipe usualmente tem apenas casos de uso em alto nível.

Mesmo assim, essas técnicas podem ser úteis se os casos de uso efetivamente já foram detalhados, ou se a estimativa de esforço deve ser refinada quando uma iteração iniciar. Por este motivo, essas técnicas são resumidas nesta seção.

Robiolo e Orosco (2008) sugerem que o texto dos casos de uso detalhados deve ser analisado e que as entidades (classes) e transações (passos) sejam contadas. O tamanho de uma aplicação em pontos de caso de uso não ajustados seria então a soma desses valores.

Braz e Vergilio (2006) propõem uma técnica chamada Pontos de Caso de Uso Difusos (*Fuzzy Use Case Points – FUSP*), na qual a contagem é baseada na complexidade de:

- Atores que participam do caso de uso.
- Pré e pós-condições do caso de uso.
- O número necessário de entidades para o caso de uso.
- A complexidade dos fluxos alternativos do caso de uso.

Assim, a medida é bastante detalhada, mas muito trabalho é necessário para aplicar esta técnica e os casos de uso devem estar expandidos para que ela funcione.

Mohagheghi, Anda e Conradi (2005) propõem uma técnica (Pontos de Caso de Uso Adaptados – *Adapted Use Case Points*) que assume que cada ator tem complexidade média e cada caso de uso é complexo no início. Mais tarde, casos de uso devem ser divididos em casos de uso menores e reclassificados como simples ou médios à medida que o processo de refinamento avança. Casos de uso estendidos e fluxos alternativos também são contados.

Kamal e Ahmed (2011) apresentam um extenso comparativo entre as técnicas mencionadas anteriormente e outros métodos de contagem.

O método *Full Use Case Size* (Ibarra; Vilain, 2010) foi proposto com o objetivo de medir o tamanho dos casos de uso. O método mede aspectos funcionais e não funcionais e, para fazer isso, precisa dos casos de uso em alto nível e de uma lista de requisitos não funcionais anotados para cada caso de uso. A medida é baseada em três componentes:

- O tipo do caso de uso.
- O número de operações de sistema (transações) e regras de negócio.
- O número de requisitos de interface.

Detalhes sobre essas técnicas podem ser encontrados nos trabalhos originais, conforme citados anteriormente.

4.3 Planejamento de projeto iterativo

O objetivo do planejamento de projeto é construir um *plano* para o projeto como um todo. Entre outras coisas, é importante que a pessoa ou grupo responsável pelo planejamento use as melhores ferramentas possíveis para avaliar a quantidade de esforço que deve ser gasta com o projeto. Esta estimativa vai levar ao cronograma e ao orçamento do projeto.

Existem vários aspectos a considerar no início. A declaração de escopo já definiu as metas do projeto e os critérios de aceitação. Além disso, a modelagem de negócio necessária já foi completada e os requisitos de alto nível foram incorporados aos casos de uso de sistema.

Com a técnica de análise de pontos de caso de uso, o esforço total para desenvolver o projeto pode ser estimado, bem como o tempo linear ideal e o tamanho médio da equipe; estas são as medidas globais para o projeto.

Agora é necessário definir a duração e o número de iterações.

4.3.1 Estimação da duração das iterações

Uma iteração começa com planejamento e termina com uma nova versão do sistema sendo entregue internamente ou para o cliente. A duração de uma iteração no UP e na maioria dos modelos ágeis usualmente varia de uma a oito semanas¹⁷, e isso depende, entre outros fatores, da complexidade do projeto e do tamanho da equipe.

Equipes pequenas, com até cinco pessoas¹⁸, podem fazer o planejamento em uma segunda-feira pela manhã, realizar o trabalho durante a semana e gerar uma entrega na sexta-feira.

Grupos com mais de 20 pessoas precisam de mais tempo para distribuir e sincronizar as atividades, já que a quantidade de trabalho é naturalmente maior. Além disso, a geração de uma entrega toma mais tempo, porque há um número maior de partes de software a serem integradas, verificadas e consertadas. Assim, neste caso, iterações com três a quatro semanas são permitidas.

Grupos com mais de 40 pessoas necessitam trabalhar em um ambiente mais formal e estruturado, com mais documentação intermediária, de forma que o fluxo de informação é naturalmente mais lento. Assim, uma iteração com seis a oito semanas é recomendável.

Outros fatores que podem afetar a duração das iterações são:

- Geração automática de código e ferramentas de software integradas ao ciclo de vida permitem iterações mais curtas.
- Uma equipe que é muito competente no UP e em técnicas de análise, design e codificação permite iterações mais curtas.
- Se a qualidade é uma preocupação crítica e consequentemente testes e revisões são intensos, então as iterações devem ser mais longas, a não ser que automação extensiva e padrões de codificação efetivos sejam usados.

4.3.2 Número de iterações

O número de iterações de um projeto depende do tempo linear dividido pela duração de cada iteração. Por exemplo, para o projeto Livir, com a equipe de iniciantes, o projeto levaria cerca de 10 meses com uma equipe de 6 pessoas. Assim, uma iteração de duas semanas deveria ser usada com um total de 20 iterações.

O número de iterações que deve ser considerado para a Elaboração e Construção depende fundamentalmente da necessidade de abordar problemas arquiteturais. Isso pode ser grosseiramente estimado pela quantidade proporcional de casos de uso complexos que devem ser estudados e desenvolvidos. Como um dos metaobjetivos da Elaboração é uma arquitetura estável, pode-se assumir que a Elaboração não terá terminado enquanto houver casos de uso complexos e com alta prioridade que precisem ser analisados. Kruchten (2003) estima que a Elaboração deva tomar cerca de 30% do tempo linear de um projeto típico e

¹⁷ Métodos ágeis preferem as durações mais curtas possíveis. Alguns métodos como XP são mais radicais ao proibirem iterações com mais de três semanas, mas outros métodos são mais relaxados em relação a isso.

¹⁸ Ou equipes maiores que sejam muito bem afinadas e tenham excelente suporte de ferramentas.

a Construção, cerca de 50% do tempo linear de um projeto típico, enquanto a Concepção e Transição tomariam cerca de 10% cada.

Outra questão que vem à mente neste ponto é se a estimação de esforço inclui a fase de Concepção ou não (porque a Concepção estará praticamente terminada quando a estimação for feita). O método COCOMO II, por exemplo, quando aplicado com o Processo Unificado, estima apenas o esforço a ser gasto na Elaboração e Construção, deixando a Concepção e a Transição para serem calculadas como uma percentagem do esforço total. Usualmente, acredita-se que a técnica de pontos de caso de uso estima o esforço total de um projeto, de seu início até a entrega final do produto. Entretanto, se a equipe usa outra interpretação, o importante é manter-se coerente com ela de um projeto para outro. Qualquer discrepância nas medidas de estimação pode ser absorvida pelo índice de produtividade da equipe.

4.3.3 Esforço por ponto de caso de uso

Para encontrar o esforço necessário para desenvolver cada tipo de caso de uso (simples, médio e complexo), o seguinte procedimento pode ser usado:

1. Tome o esforço total para o projeto (E) e divida-o pelo peso total não ajustado dos casos de uso ($UUCW$) – ignore o peso dos atores.
2. O valor resultante é o esforço estimado por ponto de caso de uso não ajustado para o projeto e equipe específicos ($E_{UCP} = E/UUCW$).
3. Finalmente, encontre o esforço para cada tipo de caso de uso multiplicando E_{UCP} por 5 para casos de uso simples, 10 para médios e 15 para complexos.

Para o exemplo corrente tem-se:

$$E_{UCP} = \frac{59,64}{220} = 0,271 \text{ funcionário-mês por ponto de caso de uso}$$

Assim:

- Casos de uso simples necessitam de $5 * 0,271 = 1,355$ funcionário-mês.
- Casos de uso médios necessitam de $10 * 0,271 = 2,710$ funcionário-mês.
- Casos de uso complexos necessitam de $15 * 0,271 = 4,065$ funcionário-mês.

Note que a equipe dos sonhos produziria valores muito menores do que estes:

$$E_{UCP} = \frac{10,42}{220} = 0,047 \text{ funcionário-mês por ponto de caso de uso}$$

Assim:

- Casos de uso simples necessitam de $5 * 0,047 = 0,235$ funcionário-mês.
- Casos de uso médios necessitam de $10 * 0,047 = 0,470$ funcionário-mês.
- Casos de uso complexos necessitam de $15 * 0,047 = 0,705$ funcionário-mês.

4.3.4 Capacidade de carga da equipe

Outra informação que pode ser calculada é a *capacidade de carga da equipe* (*Team Load Capacity* – TLC) para cada iteração. Ela é calculada como o número de membros da equipe multiplicado pelo comprimento da iteração (expresso em meses):

$$TLC = P * \text{comprimento da iteração (em meses)}$$

No exemplo corrente, considerando a equipe de novatos (seis pessoas) e iterações de duas semanas (cerca de 0,5 meses), $TLC = 6 * 0,5 = 3$. Isso significa que cada iteração para este projeto tem uma capacidade de carga da equipe de 3 funcionários-mês.

Com uma iteração de duas semanas, a equipe não seria capaz de terminar um caso de uso complexo (ele requer 4,065 funcionários-mês, como visto anteriormente). Duas opções são consideradas:

- Estender a duração das iterações para três semanas (cerca de 0,75 mês).
- Dividir casos de uso de forma que possam ser parcialmente implementados em iterações de duas semanas.

Embora desenvolvedores ágeis fossem preferir iterações mais curtas, se escolhermos estender as iterações, o tempo linear para o projeto (10 meses) dividido pela duração da iteração (0,75) resultará em 13,3 iterações, o que pode ser ajustado para 14, a não ser que haja motivos para apressar o projeto. Dessa forma, a capacidade de carga da equipe para cada iteração será de $6 * 0,75 = 4,5$ funcionários-mês e uma iteração poderia ser suficiente para lidar com um caso de uso complexo completo.

4.3.5 Definição de prioridade de casos de uso

Uma das principais técnicas para redução de risco em um projeto é lidar primeiro com os casos de uso mais complexos, especialmente se eles forem aqueles que representam os processos de negócio mais críticos, porque com eles a equipe pode aprender mais sobre o sistema do que com outros casos de uso. Casos de uso padronizados como CRUD (complexidade média) e relatórios (a menor complexidade possível) podem ser tratados mais tarde durante a Construção.

O planejamento indica quando cada caso de uso vai ser analisado, projetado e implementado. O Processo Unificado, como a maioria das técnicas ágeis, não espera que um plano de projeto geral defina quando cada caso de uso vai ser implementado. Entretanto, eles são colocados em uma lista de prioridades dinâmica e, embora ela possa ser atualizada no andar da carruagem, é possível saber quando se supõe que cada caso de uso seja completado. O planejamento de uma iteração se inicia pela escolha dos casos de uso, riscos e solicitações de mudança que devem ser tratados na iteração seguinte. Quando se planeja uma iteração, os seguintes aspectos devem ser considerados quando se escolhem quais casos de uso tratar:

- *Complexidade do caso de uso*: casos de uso de alto risco e complexidade deveriam ser tratados em primeiro lugar na maioria das vezes, a menos que condições especiais

justifiquem escolher casos de uso diferentes (por exemplo, um caso de uso relativamente simples, mas com alto risco tecnológico).

- *Dependências entre casos de uso*: casos de uso que tenham forte dependência com outros que já foram acomodados na iteração deveriam ser tratados juntos, se possível, porque isso evita fragmentar o trabalho nas classes.
- *Capacidade de carga da equipe*: o trabalho de desenvolvimento deve ser atribuído aos membros da equipe baseado no seu *TLC*.

Uma sugestão para priorização dos casos de uso do exemplo Livir apresentado na Figura 3.11 é dado na Tabela 4.7.

Tabela 4.7 Sugestão para prioridade de casos de uso para o exemplo Livir.

1. Pedir livros	12. Gerenciar livros <<crud>>
2. Pagar pedido	13. Gerenciar editoras <<crud>>
3. Enviar pedido	14. Status de pedido <<report>>
4. Registrar confirmação de entrega	15. Pedidos passados <<report>>
5. Receber livros	16. Livros vendidos por período <<report>>
6. Reenviar pedido	17. Livros disponíveis para venda <<report>>
7. Registrar devolução de pedido	18. Livros devolvidos por período <<report>>
8. Descartar livros	19. Pedidos recebidos por período <<report>>
9. Cancelar pedido	20. Entregas por período <<report>>
10. Criar/remover oferta especial	21. Livros descartados por período <<report>>
11. Gerenciar clientes <<crud>>	

A ordem desta lista é mais crítica para os casos de uso complexos, porque uma escolha ruim em termos de suas prioridades pode levar a refatorações desnecessárias nas iterações subsequentes. Considera-se que os casos de uso mais importantes para este negócio são aqueles relacionados com o processo de vendas de livros, porque isso é o que gera lucro para a empresa. Entretanto, a ordem dos casos de uso com baixo risco, tal como os *cruds* e relatórios, não é tão crítica.

4.3.6 Planejamento de fase e iterações

Com a informação obtida nas seções anteriores, é possível elaborar o plano de entregas preliminar usando-se a prioridade dos casos de uso, o esforço e a capacidade de carga da equipe.

Os objetivos de uma iteração podem ser de três tipos:

- Implementar total ou parcialmente um ou mais *casos de uso*.
- Mitigar um ou mais *riscos*, gerando ou realizando um plano de redução de probabilidade, redução de impacto ou recuperação de desastre.
- Implementar total ou parcialmente uma ou mais *mudanças* solicitadas. Como a arquitetura do sistema evolui durante as iterações, mudanças podem ser solicitadas em razão das não conformidades aos requisitos ou erros de design. Incorporar essas

mudanças no software pode ser uma das metas de uma iteração. Este tipo de meta usualmente surge apenas após a primeira iteração.

Para cada elemento (caso de uso, risco ou solicitação de mudança) deve haver uma estimação de esforço. Os elementos serão selecionados para inclusão em uma iteração ou outra, dependendo de sua prioridade (prioridades podem mudar à medida que o projeto progride – portanto, o planejamento pode mudar também). Prioridades mais altas devem ser dadas aos elementos mais complexos e arriscados, que são os que têm grande potencial para que a equipe possa aprender sobre o sistema. A sugestão é tomar em primeiro lugar:

- Casos de uso que representam os processos de negócio mais críticos, como, por exemplo, aqueles que ajudam a empresa a alcançar seus objetivos organizacionais, como obter lucro.
- Riscos de alta exposição, ou seja, aqueles que têm alto impacto e alta probabilidade de se tornarem problemas.
- Solicitações de mudança urgentes, como, por exemplo, aquelas que requerem uma refatoração da arquitetura.

Quando se consideram os elementos de mais alta prioridade, outros elementos com prioridades mais baixas, mas fortemente relacionados aos elementos de alta prioridade, podem ser adicionados à mesma iteração por conveniência. A coisa mais importante é que o esforço total atribuído à iteração não seja maior do que o valor de funcionários-mês que corresponde à capacidade de carga da equipe.

Mitigação de riscos e algumas solicitações de mudanças podem não ser passíveis de uma estimação de esforço paramétrica porque seu tamanho não pode ser mensurado em termos de pontos de caso de uso (ou qualquer outra métrica). Por exemplo, encontrar e corrigir um erro pode tomar segundos ou semanas. Nesses casos, usualmente, uma quantidade prefixada de tempo é atribuída, como, por exemplo, um membro da equipe trabalhando durante uma semana na mitigação de um dado risco. No final dessa *time box*¹⁹ previamente definida, o resultado é avaliado e se decide se a atividade deve continuar ou não.

Vamos considerar que o projeto tenha 10 meses (40 semanas) e uma duração de iteração de três semanas. Então a Concepção e a Transição deveriam tomar cerca de quatro semanas cada (10% do tempo total), a Elaboração deveria tomar cerca de 12 semanas (30% do tempo total) e ter quatro iterações de três semanas e a Construção deveria tomar cerca de 20 semanas (50% do tempo total) e ficar com cerca de sete iterações (uma delas poderia ser uma semana mais curta ou uma semana da fase de Transição poderia ser emprestada para a última iteração da Construção – a alternativa mostrada adiante).

Kroll (2004) apresenta uma sugestão para um plano de fase genérico que é adaptado para o exemplo *Livir* na Tabela 4.8. No momento, solicitações de mudança não foram incluídas porque elas aparecerão apenas quando o projeto estiver em andamento, não no

¹⁹ *Time box* ou “caixa de tempo” é um período fixo alocado para uma tarefa.

início. Assim, o plano neste caso foi feito apenas com base nas entregas dos casos de uso e na mitigação dos riscos.

Tabela 4.8 Planejamento genérico de fase para o exemplo Livir.

Fase	Iteração	Objetivo primário (riscos e casos de uso tratados)	Cronograma
Concepção	C1	Definir a visão Determinar o escopo do projeto Definir a arquitetura candidata Criar casos de uso de negócio Criar o plano de desenvolvimento de software	Semanas 1 a 4
Elaboração	E1	Instalar e testar os componentes da arquitetura Validar detalhes de requisitos Implementar casos de uso prioritários Testar a arquitetura proposta	Semanas 5 a 7
	E2-E4	Mitigar riscos arquiteturais Completar a instalação e o teste da arquitetura Fazer teste de carga em elementos arquiteturais	Semanas 8 a 16
Construção	C1-C6	Descrever e implementar casos de uso adicionais Integrar e validar o produto	Semanas 17 a 34
	C7	Descrever e implementar casos de uso adicionais Integrar e validar o produto Planejar o teste beta Planejar o manual do usuário	Semanas 34 a 37
Transição	T1	Entregar versão beta ao cliente Analisar o retorno dos usuários beta Aplicar correções Finalizar o manual do usuário e outros artefatos Fazer a entrega ao cliente	Semanas 38 a 40

Depois de concluir a fase de planejamento (usualmente ao final da Concepção), o plano para a primeira iteração da Elaboração *E1* pode ser detalhado, com os casos de uso sendo escolhidos e as atividades detalhadas e atribuídas para os desenvolvedores. Apenas quando a iteração estiver em andamento é que o planejamento da segunda iteração deve iniciar.

Um plano de iteração deve incluir o seguinte (Kroll, 2004):

1. Definição do escopo da iteração pela definição dos seus objetivos em termos de artefatos a serem produzidos, como:
 - Detalhamento e/ou implementação de casos de uso específicos.
 - Mitigação de riscos conhecidos.
 - Realização de mudanças solicitadas.
2. Criação de um plano detalhado de como a iteração vai se desdobrar.
3. Criação, identificação e gerenciamento de dependências entre tarefas.
4. Atribuição de tarefas a pessoas.

Por exemplo, considere que você esteja planejando a primeira iteração da Elaboração E1 para o projeto Livir. Considere os objetivos gerais apresentados na Tabela 4.8 e o estado corrente do sistema. Algumas metas de iteração poderiam ser definidas, tais como:

- Expandir e implementar o caso de uso 01: *Pedir livros*.
- Testar e refinar a arquitetura preliminar.
- Realizar planos de mitigação para o risco K1: *requisitos instáveis*.

Assim, o planejamento pode continuar pela definição dos entregáveis da iteração e das tarefas necessárias para produzi-los, e sua atribuição aos desenvolvedores. Equipes ágeis podem preferir produzir este plano em uma reunião de equipe, enquanto equipes seguindo RUP poderão preferir instanciar alguns dos *workflows* do RUP para determinar quais tarefas devem ser realizadas para obter os objetivos da iteração. Explicar isso em detalhes foge do escopo deste livro, mas os leitores interessados em aprender mais poderão iniciar consultando West (2002) ou Crain (2004).

4.4 O processo visto aqui

	Concepção	Elaboração
Modelagem de negócio		
Requisitos		
Análise e design		
Implementação		
Teste		
Gerenciamento de projeto	Estimar o esforço total, o tempo linear ideal e o tamanho médio da equipe para o projeto. Estimar a duração e a quantidade de iterações para cada fase. Preparar o plano de fase e o plano de iteração para a primeira iteração.	

4.5 Questões

1. Explique as diferenças entre COCOMO II, análise de pontos de função e análise de pontos de caso de uso. Quais são suas relativas vantagens e desvantagens?
2. Em análise de pontos de caso de uso, quais são as diferenças entre fatores técnicos e fatores ambientais? Por que a análise de pontos de caso de uso os considera como separados?
3. Seja E o esforço total para desenvolver um projeto e T o tempo linear ideal recomendado para o mesmo projeto, o que acontece com E quando o projeto deve ser desenvolvido em um tempo menor do que T ? O mesmo ocorre se o projeto puder ser desenvolvido em um tempo maior do que T ?
4. Como se determina a duração ideal de uma iteração para um dado projeto?
5. Como se estabelece uma lista de prioridades para casos de uso?

Casos de uso expandidos

5

TÓPICOS-CHAVE NESTE CAPÍTULO

- Fluxo principal de um caso de uso expandido
- Fluxos alternativos: tratadores de exceção e variantes
- Recomendações de escrita
- Diagramas de sequência de sistema

5.1 Introdução aos casos de uso expandidos

Como explicado no Capítulo 3, casos de uso podem ser usados para representar requisitos de sistema, entre outras coisas. Atualmente, eles são uma das abordagens mais importantes para requisitos porque permitem a construção de uma estrutura que é mais fácil de entender, comunicar e gerenciar do que as abordagens anteriores. Ivar Jacobson identifica algumas razões pelas quais casos de uso se tornaram tão populares¹:

- Um modelo de casos de uso é uma imagem que permite à equipe descrever um sistema complexo. Ele coloca em termos simples o que o sistema vai fazer pelos seus usuários.
- Um caso de uso produz valor para um usuário particular e não para uma comunidade não identificada deles.
- Casos de uso são também casos de teste; quando uma equipe termina de organizar os requisitos em casos de uso eles também já produziram a estrutura necessária para os casos de teste.
- Casos de uso são o ponto de partida para projetar experiências de usuário efetivas, tais como, por exemplo, um web site.
- Casos de uso dirigem o desenvolvimento da análise ao design, do design para o código e do código ao teste.

Frequentemente, a primeira atividade de análise nas iterações da Elaboração consiste em detalhar os casos de uso atribuídos para a iteração. O processo de detalhamento ou *expansão* de um caso de uso em uma sequência de passos corresponde ao refinamento da análise de requisitos, especialmente dos requisitos funcionais. Em cada iteração é possível expandir um ou mais casos de uso e analisar e projetar as principais estruturas necessárias para que o produto de software atenda aos requisitos, permitindo que os usuários interajam com o sistema ao seguir os passos indicados no caso de uso expandido.

Para fazer a expansão de um dado caso de uso em alto nível, é necessário ter em mãos o diagrama de casos de uso de sistema, incluindo todas as anotações, e o documento de

¹ Disponível em: <blog.ivajacobson.com/use-cases-%E2%80%93-why-successful-and-popular/>.

especificações suplementares. Também seria útil, na primeira iteração, ter uma versão do modelo conceitual preliminar que foi desenvolvido durante a Concepção. Para as iterações seguintes, será muito útil ter uma versão do modelo conceitual em evolução já refinado pelas atividades das iterações passadas.

Como a expansão dos casos de uso corresponde ao refinamento da análise de requisitos, a equipe deve conduzir um exame detalhado no processo incorporado no caso de uso *do ponto de vista do usuário*. O caso de uso deve ser descrito passo a passo: como ele é realizado e quais interações entre atores e sistema existem. Tecnologia e interfaces ainda devem ser ignoradas neste ponto. *Apenas a informação trocada entre o sistema e os atores é obrigatória nessas descrições.*

Essa descrição passo a passo, em princípio, não deve ser estruturada com desvios. Ela deve ser baseada em uma *sequência default*, ou *fluxo principal* (Seção 5.2), a qual descreve o que acontece quando tudo vai bem durante a interação do caso de uso. Este fluxo é também chamado de “*caminho feliz*” porque nele não existem erros, falhas ou exceções.

Após descrever o fluxo principal de um caso de uso, as regras de negócio ou requisitos não funcionais anotados para o caso de uso devem ser verificados e analisados do ponto de vista do usuário para descobrir o que poderia possivelmente dar errado em um dado passo, o que poderia gerar uma *exceção*. Após identificar uma possível exceção, um procedimento para corrigir o problema deve ser descrito (Seção 5.3.3). Deve haver pelo menos uma exceção para cada regra de negócio. O caso de uso é então completado com *fluxos alternativos* que lembram os *handlers* usados para tratamento de exceção em programação.

Além do que foi mencionado antes, a equipe deve verificar se o usuário pode ser capaz de executar o caso de uso de diferentes maneiras, ou seja, em diferentes *cenários* (Seção 5.3.1). Se a estrutura de execução varia muito, então variantes do fluxo principal podem ser identificadas (Seção 5.3.2).

5.2 Fluxo principal

O *fluxo principal* é a seção mais importante de um caso de uso expandido. Ele é a descrição do processo do caso de uso quando tudo dá certo, ou seja, quando nenhum fluxo alternativo ocorre.

No exemplo da livraria, um primeiro rascunho do fluxo principal do caso de uso *Pedir livros* poderia incluir o comprador identificando-se, pesquisando e selecionando livro e finalmente procedendo ao pagamento.

Em relação à identificação do usuário, um problema recorrente surge aqui: e se o usuário já tiver se identificado em um caso de uso que tenha sido executado imediatamente antes do atual? Esta é uma situação comum e precisa ser adequadamente resolvida. Como neste ponto a identificação é opcional, ela pode ser deixada como um fluxo alternativo, como explicado na Seção 5.3. Assim, assume-se para o fluxo principal desse caso de uso

que o usuário já seja conhecido pelo sistema; se não for o caso, um fluxo alternativo onde o usuário se identifica deve ser executado.

Portanto, as ações que melhor descrevem o fluxo principal do caso de uso *Pedir livros* são as seguintes: um usuário pesquisa e seleciona livros para comprar e procede ao pagamento. Entretanto, esta comunicação não é de mão única, e o sistema deve apresentar ao usuário quais são os livros disponíveis para venda, bem como outras informações importantes tais como o preço dos livros e o valor total do pedido.

A Figura 5.1 apresenta um rascunho para o fluxo principal do caso de uso *Pedir livros*, que é considerado o de mais alta prioridade para o sistema Livir (Seção 4.3.5).

No passo 1, o usuário pode escolher não fornecer nenhuma palavra-chave e, assim, no passo 2, ele receberá uma lista com todos os livros disponíveis na livraria. Por enquanto, não é necessário decidir sobre o formato da interface. Por exemplo, se a lista de livros for muito longa, talvez uma lista limitada com comandos de *scrolling* seja preferível a uma lista completa. Isso pode ser decidido quando a interface estiver sendo projetada. A interação do usuário com os comandos de *scrolling* da interface é algo que não deverá ser incluído na descrição essencial de um caso de uso.

Para executar o passo 5, é necessário que o usuário seja identificado e validado, ou senão como ele poderá retomar o pedido mais tarde? O fluxo principal aqui assume que ele já foi identificado mas se este não for o caso, uma exceção deveria ser levantada, conforme explicado na Seção 5.3.3.

Após o passo 5, assume-se que o caso de uso termina e que o carrinho de compras foi salvo (como esta é uma operação interna, não é necessário mencionar isso no caso de uso, como será explicado na Seção 5.4.6). Neste ponto, o usuário pode escolher sair do sistema e em outro momento fazer o pagamento ou proceder imediatamente para o caso de uso *Pagar pedido* (Figura 5.2).

Caso de uso 01: *Pedir livros*

1. O comprador fornece palavras-chave para pesquisar livros.
 2. O sistema gera uma lista de livros para venda que satisfaz as palavras-chave incluindo pelo menos título, autor, preço, número de páginas, editora, ISBN e imagem de capa.
 3. O comprador seleciona livros da lista e indica a quantidade desejada para cada um.
 4. O sistema gera um resumo do pedido (título, autor, quantidade, preço unitário e subtotal para cada livro) e o valor total.
 5. O comprador finaliza o pedido.
-

Figura 5.1

Primeiro esboço para o fluxo principal do caso de uso *Pedir livros*.

Caso de uso 02: *Pagar pedido*

1. O sistema gera o resumo do pedido pendente (título, autor, quantidade, preço unitário e subtotal para cada livro), bem como o valor total do pedido e uma lista dos endereços registrados para o comprador.
 2. O comprador seleciona um endereço de entrega.
 3. O sistema apresenta a taxa de entrega, a data de chegada prevista e uma lista de cartões de crédito já registrados para o comprador (bandeira e últimos quatro dígitos do número do cartão de crédito).
 4. O comprador seleciona um dos seus cartões para pagamento.
 5. O sistema envia para a respectiva operadora de cartão de crédito os seguintes dados: número do cartão, nome do titular, validade, código de segurança, valor total da compra e código da loja.
 6. A operadora de cartão de crédito aprova a venda pelo envio de um código de autorização.
 7. O sistema informa ao comprador que a compra foi aprovada e apresenta o número de registro para rastreamento do pacote.
-

Figura 5.2

Primeiro esboço do fluxo principal do caso de uso *Pagar pedido*.

Note que esses casos de uso são esboços porque alguns pontos ainda precisam ser discutidos e algumas melhorias na sua formulação precisam ser tomadas, as quais ocorrem nas seções seguintes. Por exemplo, os fluxos alternativos que são apresentados mais adiante são relacionados com possíveis exceções e variantes identificadas pela equipe. Por exemplo, o passo 3 de *Pedir livros* pode apresentar uma exceção tal como “quantidade insuficiente de livros no estoque”. Outra exceção evidente pode acontecer no passo 5 de *Pagar pedido*, no qual a operadora de cartão de crédito aprova a venda: embora o sucesso da operação neste caso seja esperado, ele pode não acontecer.

Exceções, entretanto, em vez de produzir testes condicionais no fluxo principal, devem ser ignoradas por ele e tratadas separadamente. Seria difícil entender o propósito de um caso de uso que tivesse em seu corpo uma estrutura algorítmica que considerasse todas as possíveis exceções, como o caso de uso mostrado na Figura 5.3.

O problema com o fluxo do caso de uso da Figura 5.3 que o deixa tão desnecessariamente complexo é a quantidade de “se-então” explícitos ou implícitos na sua formulação. O fluxo principal não deveria ser salpicado com condições. Ele deveria indicar o que poderia acontecer em uma situação ideal na qual toda a informação passada é correta e completa. Exceções e outras variações ao fluxo principal deveriam ser tratadas como fluxos alternativos, como será mostrado na próxima seção.

Caso de uso 01: *Pedir livros* (tratando exceções inadequadamente no fluxo principal) EVITAR!

1. Se o comprador não tem registro, ele se cadastra, indica seu nome, telefone, endereço, CPF e pelo menos um cartão de crédito. Se ele já for cadastrado, então ele se identifica, a não ser que já tenha se identificado antes.
 2. Se o comprador ainda não pagou pelo último pedido, o sistema mostra o carrinho de compras de forma que ele possa ser pago, cancelado ou retomado.
 3. O sistema apresenta as opções de livros para venda, indica pelo menos o título, autor, preço, número de páginas, editora, ISBN e imagem de capa.
 4. O comprador seleciona livros e indica a quantidade desejada de cada um. Se a quantidade desejada não está disponível no estoque, então o sistema deve pedir ao usuário para refazer o pedido considerando a quantidade disponível. Se a quantidade disponível for zero, o comprador deve ter a opção de retirar o item do pedido. Ainda é possível para o comprador dividir seu pedido em duas partes, recebendo primeiro os livros disponíveis e, depois, uma segunda remessa com os livros faltantes.
 5. ...
-

Figura 5.3

Caso de uso com exceções inadequadamente tratadas no fluxo principal.

5.3 Fluxos alternativos

Um caso de uso especifica um processo que pode ser realizado na vida real de diferentes maneiras. Duas pessoas comprando livros podem executar sequências diferentes de passos. Se as sequências forem suficientemente similares², então o mesmo fluxo deve ser usado para descrever ambas. Mas em algumas situações fluxos alternativos devem ser usados para indicar sequências que poderiam ocorrer de formas bem diferentes.

Existem basicamente dois tipos de fluxos alternativos: *variantes* (Seção 5.3.2), que indicam diferentes formas de atingir a mesma meta, e *tratadores de exceção* (Seção 5.3.3), que indicam como lidar com problemas que aparecem durante a execução do caso de uso.

Entretanto, algumas vezes as variantes são tão diferentes que elas acabam sendo consideradas casos de uso individuais. Então, variantes podem ser simples cenários (Seção 5.3.1) de um caso de uso único, ou, algumas vezes, elas podem ser a chave para descobrir novos casos de uso.

² Por exemplo, um comprador pede dois livros e outro comprador pede cinco livros: ambas as ações podem ser descritas simplesmente como “o comprador pede livros”, porque o número de livros é irrelevante para o entendimento do processo.

5.3.1 Cenários

Um caso de uso pode ser entendido como uma descrição ou especificação geral que suporta um conjunto de diferentes *cenários*. Cada cenário é uma realização particular ou *instância* de um caso de uso. Usualmente, um caso de uso tem um *cenário principal* (execução do fluxo principal) e *cenários alternativos* (execuções do fluxo principal que passam por um ou mais fluxos alternativos). Entretanto, a noção de variantes do fluxo principal pode criar certas dúvidas sobre o que deveria realmente ser um caso de uso. No exemplo da livraria, considerando o processo de compra de livros, pelo menos dois cenários poderiam ser considerados:

- Um comprador pede livros e salva o carrinho de compras para continuar comprando depois.
- Um comprador pede livros e paga por eles.

Isso é um único caso de uso com dois finais alternativos? Ou deveria ser dividido em dois casos de uso?

Ambas as opções são igualmente válidas, embora uma delas seja possivelmente mais útil do que a outra. As opções são:

- Criar mais casos de uso (um para cada cenário). Cada caso de uso tem uma estrutura simples, mas haverá um número maior de casos de uso similares.
- Agrupar os cenários similares em um único caso de uso. Haverá um número menor de casos de uso, mas cada um deles será mais complexo.

Neste ponto, é mais importante descobrir qual é a informação trocada entre atores e o sistema do que decidir se dois cenários consistem em um ou dois casos de uso. Assim, não é tão importante, com vistas ao resultado final, se as operações são descobertas ou descritas em um único caso de uso com dois cenários ou em dois casos de uso cada um com um único cenário.

A vantagem de juntar cenários alternativos em um único caso de uso é que a descrição de alguns passos não precisará ser repetida em diferentes casos de uso. Esta é a mesma vantagem que é obtida quando os atributos de diferentes classes são compartilhados por uma superclasse (Seção 6.6.1).

Entretanto, casos de uso não devem ser pensados como chamadas de procedimento: eles são descrições lineares de cenários que podem acontecer no mundo real onde um usuário está interagindo com o sistema.

5.3.2 Variantes

Admite-se em princípio que o fluxo principal seja uma sequência estrita de passos interativos: ela não é ramificada nem aninhada. Entretanto, algumas vezes poderia ser útil representar o fluxo principal de uma forma não tão linear.

Como mencionado na seção anterior, processos para pedir e pagar livros poderiam ser considerados um único caso de uso com dois cenários. Neste caso, cada cenário seria um final alternativo para o caso de uso: *Salvar carrinho de compras* ou *Finalizar*.

Essas duas possíveis finalizações podem ser representadas como *variantes* do fluxo principal de um caso de uso único.

Conforme a Figura 5.4, a seguir, o caso de uso *Comprar livros* tem dois finais possíveis: no primeiro, o carrinho é salvo de forma que o pedido possa ser retomado mais tarde, e no segundo o comprador paga e finaliza o pedido.

Caso de uso 01/02: *Comprar livros* (junção de Pedir livros e Pagar pedido)

1. O comprador fornece palavras-chave para pesquisar livros.
2. O sistema gera uma lista de livros para venda que satisfaz as palavras-chave incluindo pelo menos título, autor, preço, número de páginas, editora, ISBN e imagem de capa.
3. O comprador seleciona livros da lista e indica a quantidade desejada para cada um.
4. O sistema gera um resumo do pedido (título, autor, quantidade, preço unitário e subtotal para cada livro) e o valor total.
5. O comprador finaliza o pedido.
6. O comprador decide:
 - Salvar carrinho de compras: Variante 6a.
 - Finalizar: Variante 6b.

Variante 6a: Salvar carrinho de compras

- 6a.1. O comprador informa que o carrinho de compras deve ser salvo.
- 6a.2. O sistema informa ao comprador quantos dias o carrinho ficará disponível.

Variante 6b: Finalizar

- 6b.1. O sistema gera uma lista dos endereços registrados para o comprador.
 - 6b.2. O comprador seleciona um endereço de entrega.
 - 6b.3. O sistema apresenta a taxa de entrega, a data de chegada prevista e uma lista de cartões de crédito já registrados para o comprador (bandeira e últimos quatro dígitos do número do cartão de crédito).
 - 6b.4. O comprador seleciona um dos seus cartões para pagamento.
 - 6b.5. O sistema envia para a respectiva operadora de cartão de crédito os seguintes dados: número do cartão, nome do titular, validade, código de segurança, valor total da compra e código da loja.
 - 6b.6. A operadora de cartão de crédito aprova a venda pelo envio de um código de autorização.
 - 6b.7. O sistema informa ao comprador que a compra foi aprovada e apresenta o número de registro para rastreamento do pacote.
-

Figura 5.4

Fluxo principal de um caso de uso com variantes.

Caso de uso 01/02: *Comprar livros*

1. O comprador fornece palavras-chave para pesquisar livros.
2. O sistema gera uma lista de livros para venda que satisfaz as palavras-chave incluindo pelo menos título, autor, preço, número de páginas, editora, ISBN e imagem de capa.
3. O comprador seleciona livros da lista e indica a quantidade desejada para cada um.
4. O sistema gera um resumo do pedido (título, autor, quantidade, preço unitário e subtotal para cada livro) e o valor total.
5. O sistema gera uma lista dos endereços registrados para o comprador.
6. O comprador seleciona um endereço de entrega.
7. O sistema apresenta a taxa de entrega, a data de chegada prevista e uma lista de cartões de crédito já registrados para o comprador (bandeira e últimos quatro dígitos do número do cartão de crédito).
8. O comprador seleciona um dos seus cartões para pagamento.
9. O sistema envia para a respectiva operadora de cartão de crédito os seguintes dados: número do cartão, nome do titular, validade, código de segurança, valor total da compra e código da loja.
10. A operadora de cartão de crédito aprova a venda pelo envio de um código de autorização.
11. O sistema informa ao comprador que a compra foi aprovada e apresenta o número de registro para rastreamento do pacote.

Variante 1-7a: Salvar carrinho de compras

- 1-7a.1. O comprador informa que o carrinho de compras deve ser salvo.
 - 1-7a.2. O sistema informa ao comprador quantos dias o carrinho ficará disponível.
-

Figura 5.5

Forma alternativa de representar um caso de uso com variantes onde uma é mantida no fluxo principal e as outras são fluxos alternativos.

Salvar o carrinho para continuar a compra outro dia não é uma exceção (no sentido dado a essa palavra na Seção 5.3.3), mas uma *opção* do comprador. Isso não é uma condição que evita que o caso de uso seja concluído, porque o caso de uso ainda produz algo: o carrinho é armazenado para uso posterior. Isso é apenas uma variante do fluxo principal, e é por isso que ambas as variantes são mostradas com o mesmo *status* no fluxo³.

³ Entretanto, este estilo de variante não deve encorajar o leitor a fazer o mesmo com as exceções. Como explicado na Seção 5.3.3, no caso de exceções, existe de fato um *fluxo principal* (ou normal) e um *fluxo alternativo* (exceção), o qual deve ser descrito fora do fluxo principal; no caso de variantes, a ideia é que elas são apenas formas diferentes de realizar o fluxo principal.

Alguns analistas poderiam escolher uma das variantes para ficar no fluxo principal e outra para ser um fluxo alternativo na forma de variante. Na Figura 5.5, a variante *Finalizar* foi selecionada para ficar no fluxo principal, enquanto *Salvar carrinho de compras* foi deixada fora do fluxo principal. Uma vantagem dessa escolha é que ela permite que a variante seja associada com mais do que um passo no fluxo principal. Por exemplo, considere que a variante *Salvar carrinho de compras* pode ser executada quando o usuário estiver em qualquer dos passos entre 1 e 7. Neste caso, o código da variante poderia ser numerado como 1-7a. Se a variante puder ser executada a partir de qualquer dos passos do fluxo principal, então seu código poderia ser *a.

Este estilo, embora predominante na literatura, força o analista do caso de uso a fazer uma escolha entre as variantes, o que pode ser artificial em alguns casos. Se não há uma predominância clara entre as diferentes variantes, então um estilo como o apresentado na Figura 5.4 parece preferível, porque ele garante o mesmo *status* para cada uma das variantes.

5.3.3 Tratamento de exceções

Após descrever o fluxo principal de um caso de uso e adicionar eventuais variantes, o analista pode se concentrar na identificação das *exceções* que poderiam ocorrer e criar um fluxo alternativo para cada uma delas. Existem basicamente duas técnicas para identificar tais exceções:

1. Cada regra de negócio anotada para o caso de uso pode gerar uma exceção. Por exemplo, se o caso de uso para pedir livros tiver uma regra de negócio que estabelece que o valor de um pedido não pode ser menor do que R\$ 50,00, então quando um pedido for finalizado deve ocorrer uma exceção se ele não atingir este valor. Para lidar com essa exceção, será necessário adicionar um ou mais livros ao pedido, ou ele será cancelado.
2. Adicionalmente, o analista pode olhar para cada um dos passos do fluxo principal, especialmente aqueles que representam informação fluindo dos atores para o sistema. Nos fluxos, poderá haver possíveis dados que o sistema não pode aceitar por definição, os quais se constituem em exceções. Por exemplo, no passo onde a operadora de cartão de crédito autoriza a venda, ela poderia enviar um código de recusa – isso seria uma exceção, porque o caso de uso não poderia ser completado com sucesso neste caso. Ações corretivas devem ser tomadas (fora do fluxo principal), ou o caso de uso será abortado sem atingir seus objetivos.

Uma *exceção* (no sentido usado em ciência da computação) não é necessariamente um evento que ocorre raramente, mas um evento que se não for tratado evita que um processo continue. Uma exceção não é necessariamente algo que impede o caso de uso de iniciar, mas é normalmente algo que impede o caso de uso de ser concluído. O fato de que o cartão de crédito atingiu o limite não evita que o comprador acesse a livraria e comece

a comprar, mas isso evita que ele complete o pedido, a não ser que um cartão de crédito válido seja fornecido. Portanto, embora o caso de uso possa ser iniciado, ele não pode ser concluído até que quaisquer exceções que tenham ocorrido sejam tratadas.

A Figura 5.6 mostra uma evolução do caso de uso *Pedir livro*, agora com algumas exceções identificadas logo após o fluxo principal. Cada exceção é identificada com um código que indica a linha do fluxo principal onde a exceção se originou e uma letra que ajuda a diferenciar as exceções que podem acontecer na mesma linha. Por exemplo, se a linha 5 tem três exceções, então elas serão identificadas como 5a, 5b e 5c. Se uma exceção pode ocorrer em qualquer linha, então ela é identificada como *a, *b etc., e se ela pode ocorrer em um intervalo de linhas, tal como 3-7, então ela é identificada como 3-7a, 3-7b etc. Adicionalmente, se uma exceção pode ocorrer em linhas não adjacentes, tal como as linhas 3 e 6, então ela pode ser identificada como 3,6a ou (3,6)a. A frase que segue ao código de identificação corresponde à regra de negócio ou situação que causa a exceção. A Figura 5.6 também introduz a variante 5d que permite que um comprador continue pesquisando por outros livros em vez de necessariamente finalizar o pedido após a primeira pesquisa; isto não é uma exceção, mas apenas uma escolha do comprador.

Caso de uso 01: *Pedir livros*

1. O comprador fornece palavras-chave para pesquisar livros.
 2. O sistema gera uma lista de livros para venda que satisfaz as palavras-chave incluindo pelo menos título, autor, preço, número de páginas, editora, ISBN e imagem de capa.
 3. O comprador seleciona livros da lista e indica a quantidade desejada para cada um.
 4. O sistema gera um resumo do pedido (título, autor, quantidade, preço unitário e subtotal para cada livro) e o valor total.
 5. O comprador finaliza o pedido.
-

Exceção 3a: A quantidade solicitada é maior do que a quantidade em estoque para um ou mais livros.

Exceção 5a: O comprador ainda não se identificou.

Exceção 5b: O comprador não tem uma conta.

Exceção 5c: Nenhum livro foi selecionado para compra.

Variante 5d: O usuário deseja pesquisar mais livros.

Figura 5.6

Exemplo de caso de uso com exceções identificadas.

O número de exceções que uma equipe pode se deparar pode ser bem alto; entretanto, estamos interessados aqui em apenas umas poucas dentre elas. A maioria das exceções que poderiam ser pensadas pode ser resolvida com outros mecanismos que não a lógica do caso de uso. A chave para reduzir a complexidade do caso de uso e ainda lidar com a complexidade do sistema é reduzir o número de exceções identificadas aqui para ficar com apenas aquelas que realmente devem ser tratadas. As exceções que são de real interesse aqui são aquelas nas quais o usuário tenta enviar ao sistema algum dado que não possa ser aceito. Qualquer outro tipo de exceção provavelmente poderá ser resolvido com outros mecanismos. No exemplo da Figura 5.6:

- No passo 1, não pode haver exceções porque o sistema pode aceitar qualquer palavra-chave que o usuário forneça. Apenas pode acontecer de que para alguma escolha de palavras-chave não haja nenhum livro para apresentar. Neste caso, o comportamento do sistema deveria ser o de mostrar uma lista de livros vazia ou, talvez, mostrar os livros que mais se aproximam das palavras-chave indicadas (é o que o Google faz). Isso não é uma exceção que evita que o caso de uso continue, mas apenas um caso especial de seu comportamento.
- No passo 2, e em qualquer outro passo nos quais o sistema apresenta dados, não pode haver exceções de lógica porque no passo 2 o sistema não está tentando acomodar dados enviados pelo usuário; se alguma exceção pode ocorrer, então ela provavelmente ocorreu em um passo anterior e gerou um efeito aqui. Se o sistema mostra uma lista sem livros, isso não é uma exceção; novamente, é um caso especial de seu comportamento, e a ação do usuário seria selecionar nenhum livro no passo 3 e após isso voltar e repetir o passo 1 ou proceder para a finalização do pedido no passo 5.
- Ainda no passo 2, um analista poderia pensar “e se os cálculos envolvidos com a consulta tiverem um defeito que faz com que o sistema trave quando tentar apresentar um resultado?”. Bem, se o código tem um defeito, então ele deve ser corrigido e não incorporado à lógica do caso de uso. Não é neste tipo de exceção que estamos interessados aqui.
- Em qualquer passo, um analista poderia pensar que uma exceção poderia ocorrer se a comunicação entre o usuário e o sistema falhasse. Isso é o que os desenvolvedores usualmente entendem como uma exceção do *mundo real*. Entretanto, este também não é o tipo de exceção no qual estamos interessados aqui, porque ela não é específica à lógica do caso de uso; comunicações podem falhar a qualquer momento – então não importa qual passo do caso de uso esteja sendo executado. Adicionalmente, esta exceção é relacionada com *tecnologia* e nós estamos definindo casos de uso essenciais, onde referências para a tecnologia de implementação devem ser evitadas. Assim, este tipo de exceção deve ser tratado por mecanismos gerais do sistema e não pela lógica do caso de uso.
- No passo 3, o que acontece se o usuário indicar uma quantidade negativa? Ou se ele escrever uma letra no campo onde se espera ler a quantidade? Aqui, a ideia é

que ações ruins do usuário possam ser eliminadas por simples controles de interface (*widgets*). Por exemplo, em vez de escrever um número na caixa de texto, o usuário poderia selecionar a quantidade a partir de uma barra de rolamento que só representa quantidade positivas. Esses controles de interface são parte da tecnologia e não são mencionados nos casos de uso essenciais. Entretanto, o analista pode saber que existem formas de evitar que um usuário escolha um número negativo ou uma quantidade “xyz”, e, portanto, esta não é uma preocupação que deva ser incluída na estrutura do caso de uso. Conforme explicado no Capítulo 8, este tipo *sintático* de erro de usuário deve ser tratado pela definição dos tipos de argumentos que um sistema pode receber. Se o tipo for definido como “natural”⁴, então a equipe não deve se preocupar com quaisquer outros valores fora deste conjunto; eles simplesmente não podem ser produzidos pelo usuário. A interface não vai permitir isso.

- Entretanto, existe uma exceção lógica na linha 3 que corresponde ao usuário solicitando uma quantidade de livros que não está disponível em estoque. A quantidade de livros disponível varia de livro para livro e varia no tempo. Assim, ela não pode ser definida como um tipo estático (como, por exemplo, o intervalo [1..10]). O sistema poderia evitar que o usuário escolhesse uma quantidade maior do que a disponível, mas, neste caso, o sistema deveria ter verificado as quantidades disponíveis antes da ação do usuário.
- Finalmente, a linha 5 tem pelo menos três exceções lógicas que podem impedir o caso de uso de alcançar seus objetivos: um usuário ainda não identificado, um usuário que não possui identificação válida e um pedido sem livros.

Pela discussão anterior, pode-se concluir que na prática exceções ocorrem usualmente em passos que correspondem a informação sendo enviada *para* o sistema, porque apenas nesses casos há regras de negócio ou situações que evitam que o sistema aceite dados e continue com o processo. Muitas exceções potenciais podem ser tratadas como checagem de tipos ou como questões tecnológicas, como antes explicado. E na maioria dos casos as exceções identificadas em um passo onde o sistema está apresentando informação poderiam ter sido tratadas em um passo anterior onde o usuário estava fornecendo informação inválida para o sistema. Em resumo, *as exceções interessantes em um fluxo de caso de uso são aquelas nas quais o usuário tenta enviar informação que o sistema não pode aceitar nem impedir.*

Cada exceção deve ser *tratada* por um fluxo alternativo que corresponde a uma ramificação do fluxo principal. Este fluxo alternativo define uma sequência de passos que devem ser executados para eliminar o problema criado pela exceção. Se essa sequência de passos não puder ser executada ou se o ator não quiser realizá-la, então a única opção deixada é abortar o caso de uso, ou seja, terminá-lo sem atingir suas metas.

⁴ O conjunto {1, 2, 3, ...}.

Portanto, cada exceção identificada em um caso de uso tem um *tratador de exceção*. Um tratador de exceção tem pelo menos quatro partes:

- Um *código de identificação* que, como explicado anteriormente, consiste do número da linha (ou linhas) que originou a exceção seguindo de uma letra que identifica a exceção.
- A *exceção*, que consiste de uma frase que estabelece qual regra de negócio foi quebrada, já que na mesma linha do fluxo principal vários tipos de exceção podem ocorrer como, por exemplo, na linha 5 do caso de uso da Figura 5.6.
- *Finalização*, que é a última linha do fluxo alternativo e indica se e como o caso de uso retorna para o fluxo principal após realizar as ações corretivas.

Há cinco formas básicas de terminar uma sequência de ações corretivas:

1. *Terminar o caso de uso normalmente após as ações corretivas terem sido realizadas.* Por *default*, se nenhuma ação de finalização é mencionada, pode-se assumir que o caso de uso termina no final do fluxo alternativo. O caso de uso não retorna ao fluxo principal, mas seus objetivos são atingidos por meio do fluxo alternativo.
2. *Retornar para um passo anterior ao que causou a exceção.* Algumas vezes é necessário voltar ao início do caso de uso ou mover-se para alguns passos antes do que causou a exceção. Isso pode não ser muito comum nem muito desejável, porque implica que o usuário terá de repetir passos que já foram dados. Entretanto, em sistemas que devem obter dados em tempo real ou sistemas com transações intensivamente concorrentes, este pode ser exatamente o caso.
3. *Retornar ao passo que causou a exceção e realizá-lo novamente.* Isso é mais usual. Deve-se escolher esta opção quando o passo que causou a exceção puder eventualmente causar outras exceções, mesmo se uma delas já tiver sido tratada. Este é o caso para as exceções do passo 5 da Figura 5.6, visto que, em adição a não ter uma identificação válida, o usuário pode ainda não ter selecionado nenhum livro.
4. *Avançar para algum passo após o que causou a exceção.* Isso pode ser feito quando as ações corretivas já produziram o resultado que o passo ou conjunto de passos deveria ter feito. Entretanto, ainda é necessário verificar se novas exceções não poderiam acontecer no fluxo principal ou alternativo (ver opção anterior).
5. *Abortar o caso de uso.* Neste caso, o controle não retorna ao fluxo principal e o caso de uso não atinge seus objetivos. Se for necessário realizar algumas ações corretivas (por exemplo, remover registros temporários), isso pode ser indicado nos passos do fluxo alternativo (esta abordagem para tratamento de exceção é conhecida como *pânico organizado*).

A Figura 5.7 apresenta uma nova versão do caso de uso *Pedir livros*, agora com as ações corretivas para os fluxos alternativos definidas.

Caso de uso 01: *Pedir livros*

1. O comprador fornece palavras-chave para pesquisar livros.
 2. O sistema gera uma lista de livros para venda que satisfaz as palavras-chave incluindo pelo menos título, autor, preço, número de páginas, editora, ISBN e imagem de capa.
 3. O comprador seleciona livros da lista e indica a quantidade desejada para cada um.
 4. O sistema gera um resumo do pedido (título, autor, quantidade, preço unitário e subtotal para cada livro) e o valor total.
 5. O comprador finaliza o pedido.
-

Exceção 3a: A quantidade solicitada é maior do que a quantidade em estoque para um ou mais livros.

- 3a.1. O sistema informa as quantidades disponíveis para cada livro que foi pedido acima do estoque.
 - 3a.2. O comprador altera as quantidades desejadas para valores iguais ou menores do que o disponível no estoque.
 - 3a.3. Avança para o passo 4.
-

Exceção 5a: O comprador ainda não se identificou.

- 5a.1. O comprador fornece uma identificação válida.
 - 5a.2. Retorna ao passo 5.
-

Exceção 5b: O comprador não tem uma conta.

- 5b.1. O comprador se registra e fornece nome de usuário, senha e endereço de e-mail.
 - 5b.2. Retorna ao passo 5.
-

Exceção 5c: Nenhum livro foi selecionado para compra.

- 5c.1. Retorna ao passo 1.
-

Variante 5d: O usuário deseja pesquisar mais livros.

- 5d.1. Retorna ao passo 1.
-

Figura 5.7

Exemplo de caso de uso com tratadores de exceção.

No exemplo, cada exceção identificada e tratada é específica para o passo onde ela ocorreu. Por exemplo, a quantidade de livros acima do disponível só poderia ser detectada no passo 3, quando o usuário fornece esta informação.

É importante que apenas as exceções que são específicas aos passos sejam efetivamente incluídas no caso de uso. Caso contrário, ele poderia rapidamente se tornar desnecessariamente complexo. Exceções genéricas que podem ocorrer em qualquer

passo não devem ser tratadas como exceções aqui. Por exemplo, o ator poderia cancelar um processo em qualquer um de seus passos. Este tipo de situação é tratado de forma mais adequada por mecanismos genéricos. Para este exemplo, um mecanismo genérico de *rollback* poderia ser fornecido para o sistema. Se este tipo de exceção fosse incluído em cada caso de uso, então um número gigantesco de exceções similares poderia ser introduzido com nenhum resultado prático. Se em qualquer passo de qualquer caso de uso um usuário puder cancelar a transação, não é necessário repetir essa informação inúmeras vezes. Poderiam existir centenas ou milhares de passos no conjunto de casos de uso de um sistema. Exceções genéricas que ocorrem em um único caso de uso podem ser numeradas com a notação “*”. Caso contrário, se a exceção pode ocorrer em qualquer caso de uso ou em um número grande deles, então ela deve ser registrada como uma especificação suplementar e, portanto, será implementada por mecanismos gerais e não por passos específicos de casos de uso.

5.4 Recomendações de escrita

O estilo de escrita de um caso de uso deveria seguir um padrão do tipo “ator fornece... / sistema informa...”. Estilos como “o sistema solicita...” deveriam ser evitados porque o que deve ser representado no caso de uso é o *fluxo de informação* e não as ocasionais *demandas* que dão origem a estes fluxos; estas demandas poderão não existir em todos os cenários e a maioria delas nem sequer é necessária.

A equipe deve evitar colocar condições no fluxo principal do caso de uso. Condições como “se o usuário tem cadastro *então* o sistema apresenta...” deveriam ser evitadas. Este tipo de condição é desnecessário porque tratadores de exceção já estariam associados aos passos do fluxo principal. Assim, se uma exceção ocorrer, ela seria tratada em um fluxo alternativo.

Essas condições não deveriam ser colocadas do fluxo principal para evitar que ele se torne um fluxograma complexo em vez de uma simples e direta sequência de passos. Grandes quantidades de condições “*se/então*” poderiam deixar o caso de uso obscuro e os interessados poderiam não conseguir perceber mais qual seria o fluxo *esperado* (caminho feliz).

A única situação na qual um teste seria aceitável em um fluxo é quando existir um passo único e opcional, como, por exemplo, “se o usuário desejar, ele pode também fornecer seu número de celular”. Mas mesmo neste caso um fluxo alternativo seria uma escolha melhor.

Outra preocupação de escrita comum é que o caso de uso deve alternar entradas e saídas de informação. Em outras palavras, normalmente um caso de uso tem a forma “ator fornece..., sistema informa..., ator fornece..., sistema informa...”.

Sequências como “ator fornece..., ator fornece..., ator fornece...” ou “sistema informa..., sistema informa..., sistema informa...” devem ser evitadas a não ser que fossem justificadas.

A ideia subjacente ao padrão de alternância é reduzir a discrepância no número de passos que diferentes analistas poderiam atribuir ao mesmo caso de uso. Sem esta regra, um analista poderia escrever:

1. O comprador fornece seu nome, CPF e número de telefone.

No entanto, outro analista poderia escrever:

1. O comprador fornece seu nome.
2. O comprador fornece seu CPF.
3. O comprador fornece seu número de telefone.

A versão mais útil e correta deste ponto de vista é a primeira, porque, entre outras coisas, a segunda exige que a informação seja fornecida na ordem dos passos: após informar o CPF, o usuário não pode mais voltar atrás e corrigir seu nome, caso tenha digitado errado. Assim, a primeira versão é mais compatível com a maioria dos sistemas de informação, os quais apresentam interfaces cuja informação pode ser colocada e editada antes de ser finalmente enviada ao sistema.

Uma sequência de dois passos de entrada até poderia ser justificada se o primeiro passo pudesse causar uma exceção que evitasse que o caso de uso pudesse prosseguir. Um exemplo seria:

1. O comprador fornece identificação.
2. O comprador fornece a bandeira, número, data de validade e código de segurança de seu cartão de crédito.

Neste exemplo, quando um comprador apresenta identificação, uma exceção pode ocorrer se a identificação for inválida. Neste caso, a informação sobre o cartão de crédito não deve ser fornecida pelo usuário. O ator só poderá fornecer tal informação após o primeiro passo ter sido realizado com sucesso ou após ações corretivas de um fluxo alternativo terem sido executadas.

Mesmo neste caso, um passo intermediário de validação ainda poderia ser usado para melhorar a compreensibilidade da sequência, tal como:

1. O comprador fornece identificação.
2. O sistema informa que a identificação do comprador é válida.
3. O comprador fornece a bandeira, número, data de validade e código de segurança de seu cartão de crédito.

Finalmente, algumas mensagens de sistema podem ser evitadas no fluxo do caso de uso, porque elas não representam apresentação de informação armazenada pelo sistema. Por exemplo, o primeiro passo para o fluxo alternativo da exceção 5a na Figura 5.5 poderia ser “o sistema informa que o comprador ainda não foi identificado”. Isso não é um erro, mas o passo é de certa forma redundante no fluxo porque a exceção já estabeleceu a condição que causou a exceção. Quando uma interface for projetada para lidar com esta exceção, a equipe deverá estar preocupada com as mensagens de erro como essa. Entretanto, ela

não é necessária para o entendimento do fluxo de informação do caso de uso. Considere que se fosse necessário, então todos os tratadores de exceção de qualquer caso de uso iniciariam com a apresentação de uma mensagem, mas ela já pode ser inferida a partir da condição da exceção. Pode-se assumir por *default* que a mensagem *será mostrada* e, assim, mencioná-la toda vez não é necessário.

5.4.1 Caso de uso essencial *versus* real

Uma dúvida recorrente a respeito de casos de uso é *qual sistema* deve ser descrito? O atual? Ou o que vai ser construído? Se as operações correntes são realizadas manualmente e no futuro serão realizadas pelo computador, qual deve ser a descrição fornecida pelo caso de uso? A resposta é *nenhuma*, mas *ambas*! O caso de uso escrito para propósito de análise deve descrever a *essência* das operações e não a sua realização concreta. Assim, o analista deve sempre tentar evitar mencionar a tecnologia usada para realizar o processo e se concentrar na informação trocada entre atores e sistema. Em vez de escrever “o funcionário escreve o nome do cliente e seu endereço numa ficha de papel”, que corresponde a uma tecnologia não automatizada, ou “o funcionário preenche o nome e o endereço do cliente na tela principal”, que corresponde a uma tecnologia automatizada, o analista deveria registrar no caso de uso simplesmente “o cliente provê seu nome e endereço”. A última forma é independente de tecnologia e interface e representa, portanto, uma descrição essencial para a operação.

Todos os casos de uso usados para propósito de análise devem ser, em princípio, *essenciais*. Essencial neste contexto significa que o caso de uso é descrito em uma linguagem na qual apenas a essência das operações é apresentada, e não a sua contraparte concreta. Em outras palavras, o analista deve descrever “o que” acontece entre o usuário e o sistema sem informar “como” a interação vai acontecer. O analista não deve, portanto, quando estiver fazendo análise, tentar descrever a tecnologia de interface entre o usuário e o sistema. Isso será descrito durante a fase de design, na qual os casos de uso *reais* serão escritos após a interface real ter sido projetada. Casos de uso reais ficam presos a uma tecnologia de interface específica, enquanto casos de uso essenciais são independentes de tecnologia.

Por que casos de uso essenciais? De acordo com Ambler (2004), casos de uso reais (aqueles que mencionam tecnologia) contêm muitas hipóteses preconcebidas, frequentemente escondidas ou implícitas sobre as estratégias de implementação da tecnologia de interface. Um caso de uso essencial, no entanto, estabelece as necessidades ou intenções do usuário, e não a tecnologia concreta que pode dar suporte a essas necessidades.

Como a descrição do caso de uso neste momento é feita sem referência à tecnologia de interface, não é necessário decidir sobre como será a aparência das interfaces, mas apenas

quais informações serão trocadas entre os atores e o sistema. Referências a menus, janelas, botões e outros dispositivos gráficos devem ser evitadas no texto dos casos de uso essenciais.

Assim, como descrever casos de uso que parecem ser particularmente concretos, tais como, por exemplo, sacar dinheiro de uma máquina de saque automático? Como fazemos isso sem descrever a tecnologia? É possível. Em vez de dizer “o usuário passa o cartão magnético” pode-se dizer que “o usuário fornece identificação”. Em vez de dizer que a máquina “imprime o extrato”, pode-se dizer que a máquina “apresenta o extrato”. Assim, pela eliminação das referências à tecnologia física, sobra apenas a essência. Isso abre caminho para um design posterior baseado em tecnologia inovadora, ou seja, interfaces diferentes daquelas que se poderia imaginar em um primeiro momento. Por exemplo, o usuário poderia ser identificado pelas suas impressões digitais, o recibo poderia ser enviado por *Short Message Service* (SMS), e mesmo o dinheiro poderia ser entregue na forma de créditos em um cartão pré-pago.

É responsabilidade do analista estudar os processos atuais da empresa e produzir uma versão essencial deles na forma de casos de uso. Mais tarde, o designer vai dar uma nova forma a esses processos com o uso da tecnologia para produzir uma nova versão concreta do sistema.

5.4.2 Informação explícita

Para que os casos de uso detalhados sejam bons requisitos funcionais, é recomendado que o analista deixe explícito quais são as peças de dados individuais trocadas entre os atores e o sistema. Assim, em vez de um lacônico “o comprador fornece seus dados”, o caso de uso deveria dizer “o comprador fornece seu nome, CPF, e-mail, telefone e endereço”.

Eventualmente, um termo como “comprador” poderia ser definido externamente como significando “nome, CPF, e-mail, telefone e endereço”. Uma expressão como *comprador* = *<nome, CPF, e-mail, fone, endereço>* poderia ser escrita. Se esta definição for adequadamente registrada, por exemplo, no modelo conceitual, ou no dicionário de dados, então ela pode ser usada no texto do caso de uso para evitar ter de repetir os elementos individuais toda vez que essa informação for necessária.

5.4.3 Identificação e seleção

Um problema recorrente em casos de uso é como identificar pessoas e coisas. Por exemplo, um comprador deve ser identificado e validado para poder concluir o caso de uso *Pedir livros*. Como ele faz isso? *Username* e senha? Número do CPF? Impressões digitais? Selecionar o nome a partir de um menu ou lista predefinida? Para manter o caso de uso essencial, esta ação particular deve ser deixada genérica, e referências para possíveis formas de identificar um comprador devem ser deixadas para o design. Além disso, a comunidade

de segurança computacional já estabeleceu métodos padrão para validar de forma segura uma identidade, e estes métodos já testados deveriam ser usados em vez de se reinventar a roda. Por enquanto, o caso de uso vai simplesmente estabelecer que o usuário *forneça a identificação*, sem mencionar por quais meios essa identificação foi obtida e validada. Isso pode ser feito quando os aspectos de segurança forem alvo do design do sistema.

Outro exemplo é a seleção de livros para compra. O comprador deve submeter o ISBN do livro? Selecionar o livro de uma lista? Usar uma interface de reconhecimento de voz? Durante a expansão dos casos de uso, esses aspectos não precisam ser decididos (a menos que eles sejam efetivamente requisitos não funcionais ou suplementares). No nível essencial de um caso de uso, o comprador simplesmente *seleciona livros*. Mais tarde, o design vai apresentar algumas opções para fornecer os meios para que o usuário possa fazer esta seleção da forma mais confortável possível.

5.4.4 Passos obrigatórios

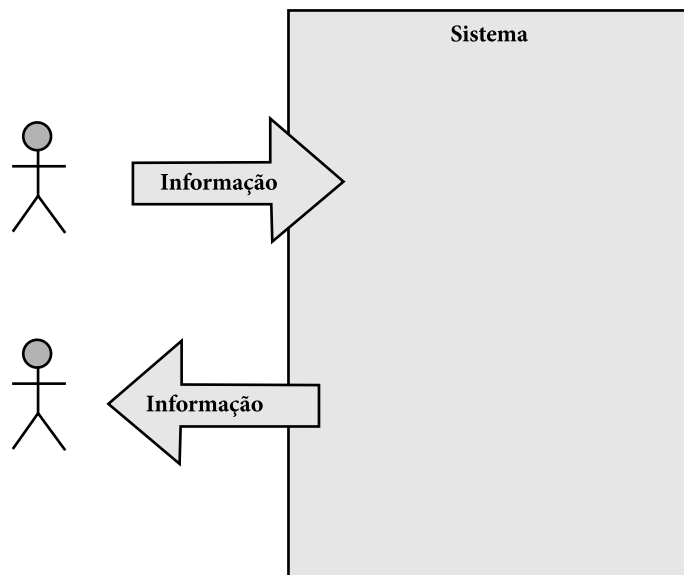
Uma das maiores dúvidas relatadas por equipes que trabalham com casos de uso refere-se a quais passos incluir no caso de uso expandido.

De fato, duas pessoas que descrevem o mesmo caso de uso, a não ser que elas tenham uma convenção claramente estabelecida, podem produzir diferentes sequências de passos. Por exemplo, em uma livraria tradicional, um analista poderia estabelecer que o funcionário pergunte ao cliente quais livros ele quer levar; mas outro analista poderia ignorar este passo e estabelecer que o comprador simplesmente pega os livros e os apresenta ao funcionário para comprar. A questão é: qual das duas versões está correta?

Ambas poderiam estar corretas, mas uma delas é mais útil. Em qualquer cenário de um caso de uso, é *obrigatório* que o comprador selecione os livros que ele pretende comprar; caso contrário, o pedido não poderia ser completado. Entretanto, é opcional para o funcionário *perguntar* pelos livros. Isso poderia acontecer apenas em alguns cenários, mas não em todos eles. Assim, selecionar livros para comprar é obrigatório, mas pedir por essa informação é opcional no fluxo do caso de uso.

Analistas são encorajados a construir versões corretas dos casos de uso, mas não se pode parar por aí: diferentes analistas deveriam construir descrições similares para o mesmo processo. Isso pode ser obtido quando o número de passos do caso de uso é minimizado.

Cada caso de uso possui *passos obrigatórios*. Estes passos incluem a informação que é passada dos atores para o sistema e do sistema para os atores (Figura 5.8). Sem um desses passos, o caso de uso pode não fazer sentido. Estes passos devem aparecer em qualquer caso de uso expandido, e sua falta implicaria que o caso de uso estaria incorreto.

**Figura 5.8**

Passos obrigatórios de casos de uso.

Outros passos tais como “pedir livros” são opcionais ou *complementares*. Eles podem ajudar a contextualizar o caso de uso, mas eles não são fundamentais, porque eles não transmitem nenhuma informação por meio da fronteira do sistema.

Assim, um caso de uso expandido para comprar livros deve conter passos que indiquem que o comprador seleciona os livros para serem comprados. O comprador poderia também informar quantas cópias de cada livro ele quer, se for o caso. O sistema deve informar ao comprador o valor total da compra. Por que esses passos são obrigatórios? Porque sem trocar essas informações seria impossível registrar adequadamente um pedido. Certamente não seria de grande ajuda se um sistema registrasse um pedido sem mencionar quais livros foram pedidos ou quanto foi gasto. Já que essa informação é tão importante para a conclusão do caso de uso, ela é considerada obrigatória para o caso de uso expandido.

Em uma descrição de caso de uso a informação não pode aparecer do nada. Ela é transmitida dos atores para o sistema e vice-versa. A falta de um passo obrigatório pode fazer com que o caso de uso pareça incompleto, como mostrado na Figura 5.9, onde um caso de uso foi descrito de forma incompleta porque uma informação obrigatória foi omitida.

Além disso, a informação contida nos passos obrigatórios dos casos de uso será usada para refinar o modelo conceitual, o qual é a base para a estrutura de classes do sistema.

Dependendo da direção na qual a informação flui, passos obrigatórios podem ser identificados como:

Caso de uso 01: *Pedir livros* (com informação obrigatória faltando) EVITAR!

1. O comprador fornece palavras-chave para pesquisar livros.
 2. O sistema gera uma lista de livros para venda que satisfaz as palavras-chave incluindo pelo menos título, autor, preço, número de páginas, editora, ISBN e imagem de capa.
 3. O sistema gera um resumo do pedido (título, autor, quantidade, preço unitário e subtotal para cada livro) e o valor total.
 4. O comprador finaliza o pedido.
-

Figura 5.9

Um exemplo de uma descrição fraca de um caso de uso na qual falta um passo obrigatório.

- *Eventos de sistema*: passos que indicam que alguma informação foi transmitida dos atores para o sistema.
- *Respostas de sistema*: passos que indicam que alguma informação foi transmitida do sistema para os atores.

Um cuidado especial deve ser tomado quando se considera as respostas de sistema. Para esses passos serem realmente obrigatórios, eles devem passar alguma informação que o sistema armazena ou acessa. Essa informação deve ser algo que em princípio os atores não tenham acesso, a não ser que façam uma consulta ao sistema.

Por exemplo, o sistema deve informar ao ator o valor total da compra no caso de uso *Pedir livros*. Caso contrário, o ator não saberá quanto tem de pagar. Essa informação poderia, talvez, ser calculada pelo comprador se ele mentalmente adicionasse os valores individuais dos livros, mas mesmo assim ele não poderia ter certeza sobre o valor total, porque informações sobre descontos, impostos e taxas teriam de ser fornecidas pelo sistema. A responsabilidade de fornecer tais informações pertence ao *sistema*.

No entanto, quando um comprador envia alguma informação ao sistema, a interface pode fornecer algum tipo de *feedback* para deixar claro que os dados foram recebidos e adequadamente processados. Entretanto, este *feedback* (normalmente uma mensagem como “ok” ou “transação completada”) não é uma resposta de sistema, porque ela não está fornecendo informações armazenadas ou derivadas sobre livros, compradores ou pedidos. Este tipo de mensagem é apenas um *feedback* de interface e, portanto, pertence ao domínio da tecnologia de interface. Como ela não consiste de informação armazenada, não deve ser considerada um passo obrigatório, mas complementar.

Pode ser interessante que os eventos e as repostas de sistema sejam explicitamente identificados nos casos de uso. Um leitor que seja iniciante em casos de uso pode querer identificar claramente os passos obrigatórios por meio da colocação de marcas [IN] para eventos e [OUT] para respostas. Essas marcas podem ser colocadas logo após o número da linha, como mostrado na Figura 5.10.

Caso de uso 01: *Pedir livros*

1. [IN] O comprador fornece palavras-chave para pesquisar livros.
 2. [OUT] O sistema gera uma lista de livros para venda que satisfaz as palavras-chave incluindo pelo menos título, autor, preço, número de páginas, editora, ISBN e imagem de capa.
 3. [IN] O comprador seleciona livros da lista e indica a quantidade desejada para cada um.
 4. [OUT] O sistema gera um resumo do pedido (título, autor, quantidade, preço unitário e subtotal para cada livro) e o valor total.
 5. [IN] O comprador finaliza o pedido.
-

Exceção 3a: A quantidade solicitada é maior do que a quantidade em estoque para um ou mais livros.

- 3a.1. [OUT] O sistema informa as quantidades disponíveis para cada livro que foi pedido acima do estoque.
 - 3a.2. [IN] O comprador altera as quantidades desejadas para valores iguais ou menores do que o disponível no estoque.
 - 3a.3. Avança para o passo 4.
-

Exceção 5a: O comprador ainda não se identificou.

- 5a.1. [IN] O comprador fornece uma identificação válida.
 - 5a.2. Retorna ao passo 5.
-

Exceção 5b: O comprador não tem uma conta.

- 5b.1. [IN] O comprador se registra e fornece nome de usuário, senha e endereço de e-mail.
 - 5b.2. Retorna ao passo 5.
-

Exceção 5c: Nenhum livro foi selecionado para compra.

- 5c.1. Retorna ao passo 1.
-

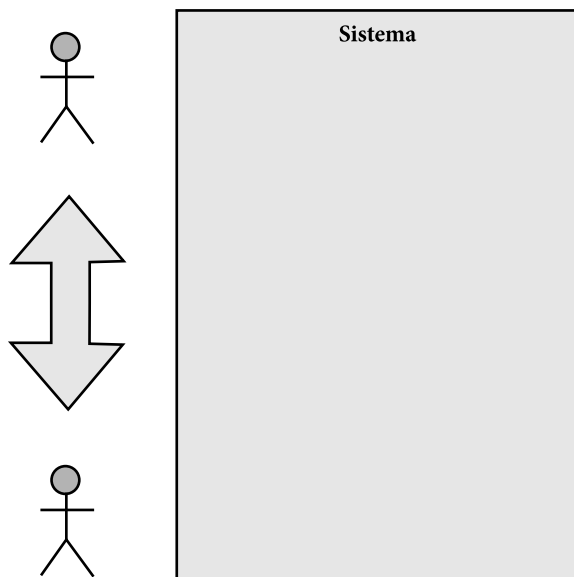
Variante 5d: O usuário deseja pesquisar mais livros.

- 5d.1. Retorna ao passo 1.
-

Figura 5.10

Um caso de uso com passos obrigatórios marcados.

Essas marcas também podem ajudar o analista a verificar se cada passo é realmente uma transação, porque nenhum passo pode ser marcado com [IN] e [OUT] ao mesmo tempo. Isso também ajuda o analista a verificar se há sequências de [IN] ou sequências de [OUT] que poderiam ser juntadas em um único passo.


Figura 5.11

Passos complementares de caso de uso.

5.4.5 Passos complementares

Outro tipo de passo de caso de uso é o passo opcional ou *complementar*, o qual não representa nenhum fluxo de informação entre os atores e o sistema. Entretanto, este tipo de passo pode ser usado algumas vezes para ajudar a entender o contexto do caso de uso.

Este tipo de passo usualmente corresponde à comunicação entre atores (comunicação que não envolve o sistema, como mostrado na Figura 5.11) ou à descrição de ações ou atitudes que também não se configuram como fluxo de informação, tais como “o comprador acessa a página da livraria”, “o comprador decide comprar livros”, “o sistema pede a identificação do comprador” ou “o funcionário pede ao comprador quais livros ele quer comprar”.

Passos complementares não são fundamentais nos casos de uso essenciais porque eles não correspondem a eventos de sistema ou retornos de sistema, já que eles não transmitem informação que atravesse a fronteira do sistema.

Alguns deles podem mesmo ser mapeados como operações de navegação durante o design da interface. Por exemplo, um caso de uso real (de design) poderia ter um passo como “o cliente seleciona uma opção”. Esta linha não corresponde a um evento de sistema, porque, nesse momento, o cliente não está passando nenhuma informação ao sistema (como nome, telefone, endereço, título do livro etc.). Esta ação corresponde simplesmente a uma mudança de estado que será implementada como navegação na interface (uma nova janela será aberta após a opção ser selecionada, por exemplo).

5.4.6 Passos impróprios

A equipe deve ter em mente que, como o caso de uso é uma descrição da interação entre atores e o sistema, ela deve evitar incluir nele quaisquer processos internos ao sistema (Figura 5.12), tais como “o sistema armazena a informação na base de dados”. Esses passos são considerados *impróprios* para a descrição do caso de uso.

O caso de uso é uma ferramenta para descrever a interação entre usuários e um sistema, não uma ferramenta para descrever processos internos. Os processos internos do sistema serão mais bem descritos durante o design com ferramentas mais adequadas, tais como diagramas de comunicação.

Na descrição do caso de uso, o analista deve se concentrar em descrever a informação que é passada dos usuários para o sistema (por exemplo, “o usuário seleciona os livros que deseja”) e do sistema para os usuários (por exemplo, “o sistema apresenta o total”). Qualquer aspecto sobre processamento interno deve ser omitido. Poderia ser dito que o sistema *apresenta* o valor total, mas não os passos tomados pelo sistema para calcular este total. Em outras palavras, o sistema deve ainda ser visto como uma caixa preta neste ponto da análise.

Se o analista quiser registrar uma fórmula ou algoritmo para fazer um determinado cálculo que é necessário para apresentar algum dado, então isso pode ser feito nas anotações do caso de uso e registrado como um requisito não funcional.

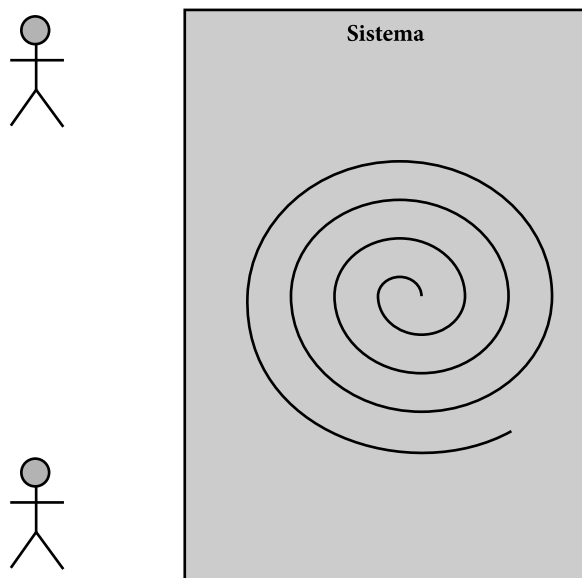


Figura 5.12

Passos impróprios para casos de uso.

Caso de uso 01: *Pedir livros* (com passos impróprios) EVITAR!

1. O comprador fornece palavras-chave para pesquisar livros.
 2. [X] O sistema verifica no banco de dados quais os livros que satisfazem o critério de pesquisa.
 3. O sistema gera uma lista de livros para venda que satisfaz as palavras-chave incluindo pelo menos título, autor, preço, número de páginas, editora, ISBN e imagem de capa.
 4. O comprador seleciona livros da lista e indica a quantidade desejada para cada um.
 5. [X] O sistema calcula o total da venda como a soma dos subtotais, os quais consistem do preço de cada livro multiplicado pela respectiva quantidade desejada.
 6. O sistema gera um resumo do pedido (título, autor, quantidade, preço unitário e subtotal para cada livro) e o valor total.
 7. O comprador finaliza o pedido.
 8. [X] O sistema marca o pedido como “finalizado” e o armazena no banco de dados.
-

Figura 5.13

Exemplo de um caso de uso com passos impróprios.

A Figura 5.13 apresenta uma situação na qual a descrição de um caso de uso vai além do recomendado já que passos impróprios foram adicionados. Os passos impróprios estão marcados com [X].

O caso de uso da Figura 5.13 inclui todos os passos obrigatórios, mas também inclui passos impróprios (passos 2, 5 e 8) que correspondem a processamento interno, não a entrada e saída de informação.

O fato de que o sistema realiza verificações, cálculos e consultas internas não afeta sua interação com o usuário. O usuário apenas precisa saber que a informação que ele enviou foi recebida pelo sistema e que ela pode ser recuperada do sistema quando necessário. Como o sistema vai garantir isso é um problema interno que será resolvido apenas durante o design e a codificação.

5.5 Casos de uso incluídos e fragmentos

É possível que dois ou mais casos de uso tenham partes coincidentes. Por exemplo, vários casos de uso poderiam incluir um fragmento *Pagamento*. Em outras situações, um caso de uso poderia incluir não apenas um fragmento, mas um caso de uso completo. A relação *include* pode ser usada apenas quando um fragmento ou caso de uso compartilhado for

Caso de uso 02: *Pagar pedido*

1. O sistema gera o resumo do pedido pendente (título, autor, quantidade, preço unitário e subtotal para cada livro), bem como o valor total do pedido e uma lista dos endereços registrados para o comprador.
 2. O comprador seleciona um endereço de entrega.
 3. O sistema apresenta a taxa de entrega, a data de chegada prevista.
 4. *Include* fragmento 02a: *Pagamento*.
 5. O sistema informa ao comprador que a compra foi aprovada e apresenta o número de registro para rastreamento do pacote.
-

Figura 5.14

Um caso de uso que invoca um fragmento incluído.

Fragmento 02a: *Pagamento*

1. O sistema apresenta a lista de cartões de crédito já registrados para o comprador (bandeira e últimos quatro dígitos do número do cartão de crédito).
 2. O comprador seleciona um dos seus cartões para pagamento.
 3. O sistema envia para a respectiva operadora de cartão de crédito os seguintes dados: número do cartão, nome do titular, validade, código de segurança, valor total da compra e código da loja.
 4. A operadora de cartão de crédito aprova a venda pelo envio de um código de autorização.
-

Figura 5.15

Fragmento de caso de uso que pode ser incluído em mais do que um caso de uso.

significativo; ela não deve ser usada quando apenas um passo do caso de uso for compartilhado. Um caso de uso ou fragmento incluído pode ser invocado em um fluxo pelo uso da expressão “include”. Suponha que *Pagamento* é um fragmento que aparece em mais do que um caso de uso no sistema. Este fragmento poderia então ser chamado por qualquer caso de uso, como mostrado na Figura 5.14.

O fragmento em si pode ser definido como mostrado na Figura 5.15.

Note que o fragmento não é um caso de uso completo neste exemplo. Ele não pode ser executado isoladamente por um usuário; ele apenas pode ser executado como parte de outro caso de uso.

Certo cuidado deve ser tomado aqui. Em primeiro lugar, *casos de uso não são procedimentos como os usados nas linguagens de programação*; eles são descrições lineares de processos do mundo real. Analistas não são encorajados a estruturar casos de uso da mesma forma que programadores fazem com programas. Casos de uso incluídos ou fragmentos podem ser usados *apenas* quando a situação realmente o justificar.

Além disso, o analista deve manter em mente que o objetivo de um caso de uso expandido na análise é ajudar a entender a natureza das interações entre o sistema e seus atores e não estruturar um programa de computador para simular esta interação. Assim, os casos de uso ou fragmentos incluídos devem ser usados com muito cuidado.

5.6 Expansão de casos de uso estereotipados

Casos de uso estereotipados ou *padronizados* apresentam risco médio ou baixo para o processo de desenvolvimento de software porque a estrutura de seus fluxos já é conhecida de antemão. A indicação dos estereótipos <<crud>> ou <<report>>, se adequadamente documentada, vai passar aos programadores uma ideia clara a respeito daquilo que deve ser implementado.

Idealmente, se os requisitos funcionais são suficientemente detalhados, casos de uso estereotipados nem sequer precisam ser expandidos. Por exemplo, se a especificação de um relatório já apresenta toda a informação que ele recebe e retorna, então não é necessário escrever um caso de uso detalhado para ele, porque isso seria apenas repetir informações que o analista já tem.

Uma preocupação que deve ser mantida em mente durante a eliciação de requisitos é se um caso de uso que foi identificado realmente pertence a um padrão. Pode ser necessário realizar algum esforço de análise sobre um caso de uso antes de decidir se ele realmente pertence a um padrão⁵. Por exemplo, relatórios nunca devem alterar dados. Se o caso de uso apresenta dados, mas também os altera, então ele não deve ser definido como um caso de uso padrão de relatório. Ele poderia ser uma variação daquele padrão ou uma estrutura completamente nova que não pertence a nenhum padrão conhecido.

No caso de CRUDs, o que pode ser dito é que nem toda entidade do modelo conceitual é passível de ser gerenciada pelo padrão CRUD. Espera-se que os CRUDs gerenciem apenas as entidades mais simples. Se eles se tornam mais complexos, então um conjunto de casos de uso que não pertencem a este padrão é necessário para descrever como o usuário trabalha com estas entidades.

Por exemplo, um *endereço* ou *cartão de crédito* são entidades bem simples que poderiam ser facilmente gerenciadas por um caso de uso CRUD. No entanto, um *pedido* é uma das entidades mais complexas do sistema de livraria. Pedidos devem ser emitidos,

⁵ Há alguns anos, o autor participou de um projeto no qual um caso de uso que parecia ser um simples CRUD à primeira vista revelou-se como o caso de uso mais complexo de todo o sistema ao final.

cancelados, completados, enviados, reenviados, recebidos de volta etc. Isso impede que eles sejam gerenciados por operações tão simples quando as do CRUD.

No meio do caminho existem conceitos que a equipe vai ter de gastar algum tempo para decidir se podem ser adequadamente gerenciados como CRUD ou não. Por exemplo, gerenciar clientes seria um CRUD ou não? Algum tempo pode ser gasto com a análise até que a equipe possa chegar a uma conclusão. A Seção 5.6.2 faz essa análise e mostra que esse caso pode ser considerado uma variação do padrão CRUD.

Os modelos de expansão para casos de uso padronizados apresentados nas subseções seguintes servem como referência para o leitor. Eles são apresentados na forma de template de maneira que analistas possam usá-los para escrever suas próprias versões dos casos de uso padronizados, ou mesmo trabalhar em novos padrões ou subpadrões.

5.6.1 Relatório expandido

Um relatório é um caso de uso muito simples que consiste em um único acesso e cálculos sobre dados gerenciados por um sistema. A informação enviada pelo ator consiste nos argumentos para seleção, filtragem, ordenação, agrupamento etc. dos dados. A Figura 5.16 apresenta um formato geral para um caso de uso de relatório expandido.

A Figura 5.17 apresenta um exemplo concreto de um relatório, isto é, uma instanciação do template aplicado ao exemplo Livir.

Template de Caso de Uso: Relatório de... <<report>>

1. O usuário fornece...
 2. O sistema apresenta..., agrupados por..., ordenados por...
-

Figura 5.16

Um template para o caso de uso relatório expandido.

Caso de Uso 18: Relatório de pedidos devolvidos por período <<report>>

1. O gerente de depósito fornece as datas inicial e final.
 2. O sistema apresenta uma lista de pedidos devolvidos no período incluindo a data da devolução, o número do pedido, a razão da devolução e uma lista dos livros devolvidos (título, quantidade e preço). A lista de pedidos é ordenada pela data e a lista de livros devolvidos é ordenada pelo título.
-

Figura 5.17

Um exemplo de um relatório expandido.

Pelas razões explicadas na Seção 5.3.3, este tipo de caso de uso não tem fluxos alternativos (caso contrário, ele não seria um relatório, mas um tipo diferente de caso de uso). O usuário simplesmente envia os argumentos e recebe um conjunto de dados organizado em resposta. Se o usuário passar argumentos errados (por exemplo, data incorreta), ele vai receber a resposta apropriada para os dados errados. Por exemplo, se o usuário passar por engano um período no qual não houve devoluções, então ele vai receber uma lista vazia, a qual corresponde à resposta certa para os argumentos passados. Neste sentido, portanto, este tipo de caso de uso não deve ter exceções.

Entretanto, a equipe deve ainda assegurar que o caso de uso é *realmente* um relatório antes de enviá-lo para o design e a codificação, porque ele ainda poderia ser uma variação do padrão.

5.6.2 CRUD expandido

Da mesma forma que os relatórios, os CRUD também são casos de uso que seguem um padrão conhecido. Este tipo de caso de uso consiste na execução opcional de uma dentre quatro opções possíveis: inserir, consultar, atualizar e excluir.

Este tipo de caso de uso tem um fluxo principal com quatro variantes, porque nenhuma das operações pode ser considerada “caminho feliz” ou “exceção”. Elas são operações distintas agrupadas mais por conveniência, já que estão associadas à mesma entidade, do que por similaridade entre seus fluxos. Portanto, o fluxo principal de um CRUD é simplesmente uma decisão do usuário sobre qual operação ele vai executar. A Figura 5.18 apresenta um possível template para este tipo de caso de uso.

Observe que, neste template de caso de uso, a variante 1c (*atualizar*) inclui os passos da variante 1b (*consultar*).

O tratador de exceção 1d.1a usa uma estratégia específica que consiste em evitar que o usuário remova um objeto quando alguma regra impede isso. Na Seção 8.5.4, este assunto é revisto e outras estratégias para lidar com esta exceção são discutidas.

A Figura 5.19 apresenta um exemplo de expansão de um caso de uso CRUD típico.

Template de caso de uso: Gerenciar... <<crud>>

1. O usuário escolhe:
 - Inserir: Variante 1a.
 - Consultar: Variante 1b.
 - Atualizar: Variante 1c.
 - Excluir: Variante 1d.
-

Variante 1a: Inserir

1a.1. O usuário fornece...

Variante 1b: Consultar

1b.1. O usuário identifica/seleciona um...

1b.2. O sistema apresenta... do elemento identificado.

Variante 1c: Atualizar

1c.1. *Include* Variante 1b: *Consultar*

1c.2. Usuário fornece novos valores para...

Variante 1d: Remover

1d.1. Usuário identifica/seleciona um... para ser removido.

Exceção 1a.1a: Inserção viola regra de negócio...

1a.1a.1. O sistema informa que a regra... impede a inserção.

1a.1a.2. Retorna ao passo 1a.1.

Exceção 1c.2a: Atualização viola regra de negócio...

1c.2a.1. O sistema informa que a regra... impede a inserção.

1c.2a.2. Retorna ao passo 1c.2.

Exceção 1d.1a: Exclusão viola regra de negócio ou regra estrutural...

1d.1a.1. O sistema informa que a regra... impede a remoção.

1d.1a.2. Retorna ao passo 1d.1.

Figura 5.18

Um template para um caso de uso CRUD expandido.

Caso de uso 13: Gerenciar editoras <<crud>>

1. O gerente de compras escolhe:
 - Inserir editora: Variante 1a.
 - Consultar editora: Variante 1b.
 - Atualizar editora: Variante 1c.
 - Excluir editora: Variante 1d.
-

Variante 1a: Inserir editora

- 1a.1. O gerente de compras fornece o nome, código de identificação, endereço e e-mail da editora.
-

Variante 1b: Consultar editora

- 1b.1. O gerente de compras identifica uma editora.
 - 1b.2. O sistema apresenta o nome, código de identificação, endereço e e-mail da editora.
-

Variante 1c: Atualizar editora

- 1c.1. *Include Variante 1b: Consultar editora*
 - 1c.2. O gerente de compras fornece novos valores para nome, código de identificação, endereço e e-mail da editora.
-

Variante 1d: Remover editora

- 1d.1. O gerente de compras identifica uma editora para ser removida.
-

Exceção (1a.1,1c.2)a: Código de identificação fornecido já existe.

- (1a.1,1c.2)a.1. O sistema informa que já existe uma editora com o código de identificação informado.
 - (1a.1,1c.2)a.2. Retorna ao passo que causou a exceção.
-

Exceção 1d.1a: Existem livros associados à editora.

- 1d.1a.1. O sistema informa que não é possível remover uma editora com livros associados.
 - 1d.1a.2. Retorna ao passo 1d.1.
-

Figura 5.19

Um exemplo de caso de uso CRUD expandido.

Neste exemplo, duas regras de negócio foram usadas para identificar exceções:

- O código de identificação para editoras deve ser único, o que pode ser verificado tanto quando uma nova editora é inserida quanto quando uma editora é atualizada. Isso produz exceções nos passos 1a.1 e 1c.2, as quais foram agrupadas em uma única exceção identificada como (1a.1,1c.2)a.

- Se a editora já tem pelo menos um livro associado a ela (e esta associação é obrigatória para os livros, isto é, cada livro deve estar ligado a uma editora), então ela não pode ser removida. Esta é uma regra estrutural que deve aparecer explicitamente no modelo conceitual como um papel obrigatório do livro para a editora (Seção 6.4.1).

A Figura 5.20 apresenta o caso de uso CRUD *Gerenciar compradores*, o qual é uma variação do padrão. Um comprador pode consultar e modificar apenas os seus próprios dados. Assim, o caso de uso deve ser adaptado para esta situação. No entanto, pode ser desejável que um gerente do sistema possa acessar dados de qualquer comprador. Para este gerente, o caso de uso seria um CRUD regular. Haveria dois casos de uso a serem criados aqui? Um para o comprador gerenciar seus próprios dados e outro para o gerente gerenciar os dados de todos? Talvez... Mas parece que esta abordagem seria um tanto repetitiva. Para piorar ainda mais as coisas, imagine que é possível que um gerente regional acesse dados apenas dos usuários da sua região. Seria este um terceiro caso de uso? Isso não parece ser uma boa solução.

Para resolver esse problema de uma maneira elegante, o caso de uso pode ser parametrizado, indicando que o usuário só pode acessar dados de compradores para os quais ele esteja autorizado a fazer. Quando compradores estão sendo gerenciados, o que deve ser verificado por uma dada regra de negócio é se o usuário realmente tem permissão para acessar ou modificar os dados do comprador que ele está tentando acessar ou modificar. Na Figura 5.20, a verificação dessa permissão é realizada pelo tratador de exceções (1b.1,1d.1) a: *Acesso não autorizado*.

Note que o caso de uso da Figura 5.20 é uma variação de CRUD que poderia ser chamada de *CRUD com restrição de acesso*, porque o acesso aos dados depende de quem é o usuário.

Existem outras variações de CRUD, tais como:

- CRUDL: Incluindo uma variante para listar os elementos existentes. Assim, em vez de simplesmente identificar um objeto para consultar, atualizar ou remover, o usuário pode selecionar ele de uma lista.
- SCRUD: Incluindo uma variante para pesquisar elementos.
- RUD: no qual elementos podem ser consultados, atualizados e removidos, mas não criados.
- CRU: no qual elementos podem ser criados, consultados e atualizados, mas não removidos.

Muitas outras variações podem ser criadas dependendo da necessidade e do custo/benefício de ter mais padrões para lembrar.

Existe uma forma mais simples de se definir um caso de uso baseado em padrões, se ele se encaixar perfeitamente no padrão. Esta forma consiste em simplesmente listar os campos a serem preenchidos e as regras de negócio a serem observadas. Para o resto, o template é aplicado. A Figura 5.21 mostra um exemplo.

Caso de uso 11: Gerenciar compradores <<crud>>

1. O usuário escolhe:
 - Inserir comprador: Variante 1a.
 - Consultar comprador: Variante 1b.
 - Atualizar comprador: Variante 1c.
 - Excluir comprador: Variante 1d.

Variante 1a: Inserir comprador

- 1a.1. O usuário fornece o nome, CPF, endereço, telefone e e-mail do comprador.

Variante 1b: Consultar comprador

- 1b.1. O usuário identifica um comprador.
- 1b.2. O sistema apresenta o nome, CPF, endereço, telefone e e-mail do comprador.

Variante 1c: Atualizar comprador

- 1c.1. *Include* Variante 1b: *Consultar comprador*
- 1c.2. O usuário fornece novos valores para nome, CPF, endereço, telefone e e-mail do comprador.

Variante 1d: Remover comprador

- 1d.1. O usuário identifica um comprador para ser removido.

Exceção (1a.1,1c.2)a: CPF fornecido já existe.

- (1a.1,1c.2)a.1. O sistema informa que já existe um comprador com o CPF informado.
- (1a.1,1c.2)a.2. Retorna ao passo que causou a exceção.

Exceção (1b.1,1d.1)a: Acesso não autorizado.

- (1b.1,1d.1)a.1. O sistema informa o usuário que o acesso é negado.
- (1b.1,1d.1)a.2. Retorna ao passo que causou a exceção.

Exceção 1d.1b: Existem pedidos associados ao comprador.

- 1d.1b.1: O sistema informa que não é possível remover um comprador com pedidos associados a ele.
 - 1d.1b.2: Retorna ao passo 1d.1.
-

Figura 5.20

Um exemplo de caso de uso CRUD expandido com restrições de acesso.

Caso de Uso 12: Gerenciar livros <<crud>>

Campos:

- ISBN, título, autor, número de páginas, editora, preço.

Regras para inserção e atualização:

- O ISBN é único, ou seja, diferentes livros não podem ter o mesmo número de ISBN.

Regras para remoção:

- Um livro com cópias já vendidas não pode ser removido.
-

Figura 5.21

Exemplo de expansão rápida de um caso de uso CRUD.

Com a informação dada na Figura 5.21 e o template da Figura 5.18, é possível implementar o caso de uso sem ter que fazer sua expansão completa.

5.7 Outras seções de um caso de uso expandido

Desde que o conceito de caso de uso foi criado (Jacobson; Christenson; Jonsson; Övergaard, 1992), diferentes formatos para ele tem sido propostos. Cada proposta inclui diferentes elementos. As seções de “fluxo principal” e “fluxos alternativos” são fundamentais para qualquer descrição de caso de uso detalhado. Entretanto, outras seções podem ser incluídas se o analista sentir necessidade delas. Algumas das mais populares são apresentadas nas subseções seguintes.

5.7.1 Interessados

Frequentemente existem cenários nos quais grupos que não são atores podem ser relevantes a um caso de uso. Outros setores de uma empresa podem ter algum interesse no caso de uso. Por exemplo, no caso de uso *Pedir livros*, o único ator é o comprador. Mas os resultados desse caso de uso podem interessar aos departamentos de estoque e financeiro, porque os resultados de uma venda podem afetar tanto a quantidade de livros no estoque quanto a quantidade de dinheiro disponível ou para ser recebida. Assim, mesmo que esses departamentos não sejam participantes do caso de uso, eles podem ser listados como *interessados*.

A utilidade da lista de interessados para um caso de uso é que este deve *satisfazer todos os interessados*. Assim, a documentação será útil para lembrar ao analista que ele deve procurar informação que precisa ser armazenada, processada ou transmitida de forma que estes interessados sejam satisfeitos.

5.7.2 Precondições

Por definição, precondições são fatos considerados verdadeiros antes que um caso de uso inicie. Precondições não devem ser confundidas com exceções: exceções só podem ser detectadas após o caso de uso iniciar. Exceções são detectadas durante um caso de uso porque na maioria das vezes não é possível verificar se as condições são verdadeiras ou não antes do caso de uso iniciar; por exemplo, não é possível saber se o comprador tem crédito suficiente para pagar pelo pedido antes que o caso de uso *Pedir livros* seja iniciado, porque o valor do pedido ainda não pode ser conhecido naquele momento; portanto, trata-se de uma *exceção*. Entretanto, é possível admitir que apenas um livro que foi vendido e enviado possa ser devolvido. Assim, a venda e o envio do livro são precondições para sua devolução.

Já que as precondições são aceitas como fatos verdadeiros antes do caso de uso iniciar, elas não são verificadas durante o caso de uso. Embora possa haver outras interpretações na literatura, este tipo de precondições não gera exceções. Se elas fossem falsas, teria sido impossível iniciar o caso de uso (caso contrário, elas seriam exceções, não precondições).

5.7.3 Pós-condições de sucesso

Pós-condições de sucesso estabelecem os resultados de um caso de uso, ou seja, o que será verdadeiro após o caso de uso ser executado. Por exemplo, o caso de uso *Pedir livros* pode ter como pós-condição de sucesso o seguinte resultado: “o pedido foi registrado pelo sistema”.

5.7.4 Pontos em aberto

Algumas vezes, sem a presença do cliente, a equipe pode não conseguir decidir-se sobre algum assunto que pode depender de políticas da empresa. Por exemplo, o comprador pode pagar em prestações? Existem ofertas especiais para compradores que compram acima de certas quantidades?

Se o cliente não estiver disponível imediatamente, essas dúvidas podem ser registradas na seção “pontos em aberto” do caso de uso para serem resolvidas o mais rápido possível.

No final das atividades de análise de uma iteração, espera-se que *todos* os pontos em aberto tenham sido resolvidos e incorporados na descrição do caso de uso.

5.8 Diagramas de sequência de sistema

Entre outras coisas, o texto dos casos de uso expandidos pode ser usado como:

- Uma fonte de informação para descobrir os conceitos e atributos que são necessários para refinar o modelo conceitual (Seção 6.8).
- Uma fonte de informação para descobrir *operações de sistema*, as quais consistem de métodos que encapsulam a camada de domínio do sistema em relação à interface (Capítulo 8).

Existem dois tipos de operações de sistema:

- *Comandos de sistema*, os quais alteram dados e não podem retornar dados.
- *Consultas de sistema*, as quais retornam dados e não podem alterar dados.

Essa divisão segue o princípio de engenharia de software conhecido como *Separação Comando-Consulta* ou CQS (*Command-Query Separation*) (Meyer, 1988), o qual viabiliza código mais reusável e manutenível. Uma consulta que altere dados teria menos coesão do que uma consulta sem efeitos colaterais e um comando sem retorno.

Comandos de sistema são métodos que são ativados por um *evento de sistema*, ou seja, como uma reação a uma ação do usuário. Comandos de sistema, por definição, implementam um fluxo de informação de fora para dentro do sistema. Portanto, comandos de sistema atualizam a informação que é gerenciada pelo sistema.

Consultas de sistema são métodos que correspondem à simples verificação de informação já armazenada. Essa informação pode ser apresentada exatamente como ela existe ou ainda ser modificada por operações lógicas ou aritméticas (tais como *soma*, *média* etc.). Por definição, uma consulta de sistema não pode ser responsável pela inserção de informação nova no sistema. Ela também não pode ser responsável pela atualização ou remoção de informações do sistema: ela só pode ler a informação.

O conjunto de operações de sistema consiste na funcionalidade total do sistema, ou seja, o conjunto de todas as funções que podem ser executadas por um usuário com o sistema.

Casos de uso são excelentes fontes para descobrir operações de sistema. Comandos de sistema são basicamente associados a passos nos quais o usuário envia alguma informação para o sistema (aqueles marcados com *[IN]*). Consultas de sistema são associadas aos passos dos casos de uso nos quais o sistema envia informação aos atores (aqueles marcados com *[OUT]*).

5.8.1 Elementos de um diagrama de sequência

Um dos diagramas UML que pode ser particularmente útil para representar a sequência de eventos e repostas em um caso de uso é o *diagrama de sequência*. Quando um diagrama de sequência é usado para representar um caso de uso, ele pode ser chamado de *diagrama de sequência de sistema* (Larman, 2004). O diagrama de sequência tem elementos que são instâncias de atores e outros componentes do sistema. Na primeira versão do diagrama de sequência de sistema, apenas os atores e a interface do sistema (ou camada de aplicação) são representados (Figura 5.22).

As atividades relacionadas à análise de casos de uso ainda não lidam com os objetos que são internos ao sistema. Assim, o sistema deve ser representado por um único objeto: uma caixa preta. Nesse caso, ele pode ser representado pelo símbolo de interface , como proposto por Jacobson *et al.* (1992). Um ator pode comunicar-se com um sistema apenas por meio dessa interface. Mensagens do ator nunca poderiam alcançar os objetos, como mostrado na Figura 5.23.

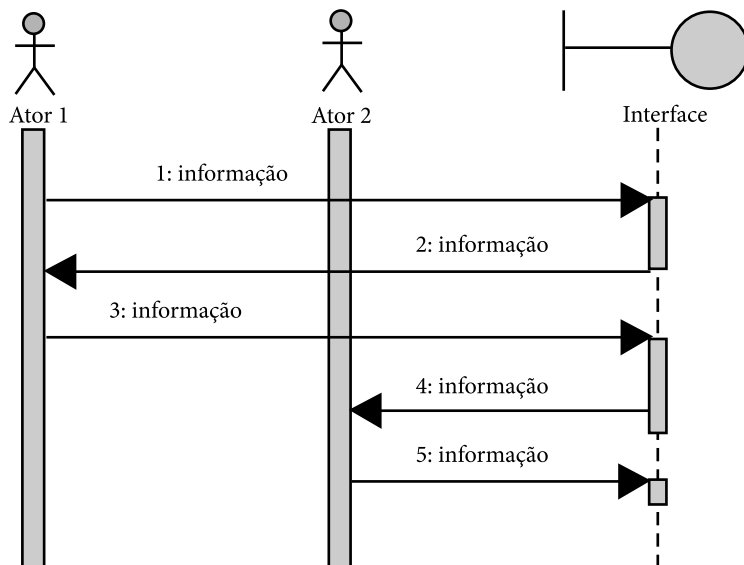


Figura 5.22

Diagrama de sequência de sistema.

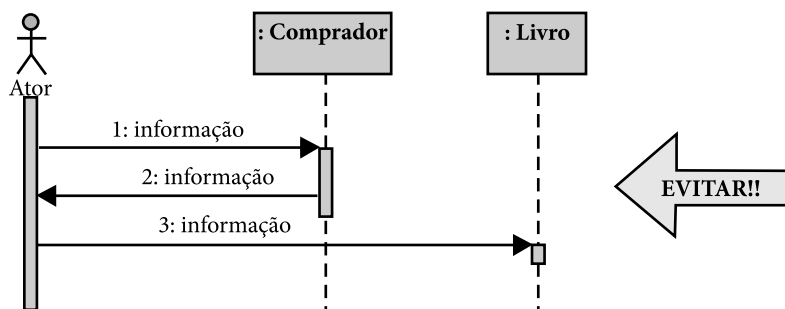


Figura 5.23

Um exemplo de um diagrama de sequência de sistema inadequado, no qual atores se comunicam diretamente com os objetos.

Atores, interfaces e outros elementos do diagrama de sequência tem uma linha de vida, representada por uma linha vertical, na qual os eventos podem acontecer. Quando a linha é tracejada, o ator ou sistema está inativo. Quando a linha está sólida, o elemento está ativo (processando ou aguardando pelo resultado de uma operação sendo realizada por outro elemento). Atores humanos são considerados sempre ativos.

As setas horizontais representam o fluxo de informação. Há três tipos de fluxos de informação no diagrama:

- *Entre os atores:* a comunicação dos atores entre si, correspondendo aos passos complementares do caso de uso expandido.

- *Dos atores para o sistema:* correspondendo aos *eventos de sistema*, ou seja, passos que poderiam ser marcados com [IN] no caso de uso expandido.
- *Do sistema para os atores:* correspondendo às *respostas de sistema*, ou seja, passos que poderiam ser marcados com [OUT] no caso de uso expandido.

A troca de informações entre os atores não pertence ao escopo do sistema, mas ela pode ser útil para ilustrar como a informação é trocada de ator para ator até chegar ao sistema.

Uma coisa a manter em mente quando esses diagramas estão sendo construídos é que *a informação não é criada durante esse processo; ela é apenas transferida dos atores para o sistema e vice-versa*. O ator usualmente tem alguma informação que deve ser passada para o sistema para que o processo possa ser realizado. Para que um pedido de livros possa ser processado, o comprador deve informar ao sistema quem ele é e quais livros ele deseja. No início, o comprador é o único que detém tal informação; o sistema tem o registro de todos os compradores e livros, mas não é capaz de saber quem é o comprador específico até que ele se identifique.

O diagrama de sequência pode ser construído para o fluxo principal de um caso de uso e completado com fluxos alternativos. Opcionalmente, diagramas de sequência distintos poderiam ser construídos para cada fluxo. Entretanto, a coisa mais importante no momento é saber *qual* informação é trocada entre os atores e o sistema. O analista deve construir um catálogo com todos os comandos e consultas de sistema (que são explicados nas próximas seções) que forem identificados nos fluxos principal e alternativos do caso de uso. Esta informação será usada mais tarde para definir contratos que indicam como o sistema recupera e transforma a informação (Capítulo 8).

5.8.2 Casos de uso expandidos como diagramas de sequência de sistema

O diagrama de sequência de sistema é uma ferramenta para obter uma descrição mais formal e detalhada de um caso de uso. Adicionalmente, o desenvolvimento de um diagrama de sequência de sistema permite fazer a conexão entre a análise de requisitos (o caso de uso expandido) e o design do software (as operações de sistema que serão implementadas).

A construção do diagrama de sequência de sistema pode ser feita em duas etapas:

1. Representar os passos do caso de uso como trocas de *informação* entre *atores* e a *interface* do sistema.
2. Representar operações de sistema como *chamadas de métodos* entre a *interface* e o *controlador-fachada*⁶, que encapsula a camada de domínio do sistema.

O primeiro passo é simples: para cada entrada de caso de uso (um passo que poderia ser marcado com [IN]) existe um evento de sistema equivalente no qual um ator envia informação para a interface, e para cada saída de caso de uso (um passo que poderia ser marcado com [OUT]), existe uma resposta de sistema equivalente na qual um ator recebe informação do sistema (Figuras 5.24 e 5.25).

⁶ Um controlador-fachada fornece “uma interface unificada para um conjunto de interfaces em um subsistema. Fachada define uma interface de nível mais alto que torna um subsistema mais fácil de usar” (Gamma; Helm; Johnson; Vlissides, 1995 – tradução do autor).

Caso de uso 01: *Pedir livros*

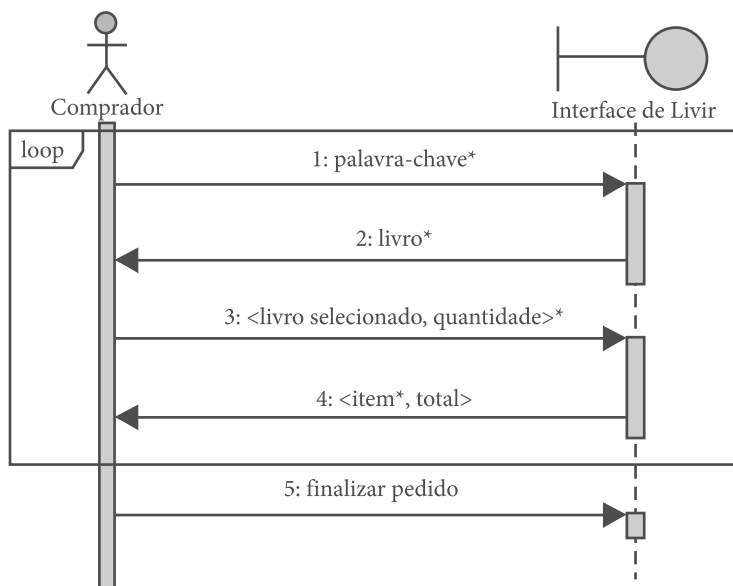
1. O comprador fornece palavras-chave para pesquisar livros.
2. O sistema gera uma lista de livros para venda que satisfaz as palavras-chave incluindo pelo menos título, autor, preço, número de páginas, editora, ISBN e imagem de capa.
3. O comprador seleciona livros da lista e indica a quantidade desejada para cada um.
4. O sistema gera um resumo do pedido (título, autor, quantidade, preço unitário e subtotal para cada livro) e o valor total.
5. O comprador finaliza o pedido.

Variante 5d: O comprador decide continuar comprando

5d.1. Retorna ao passo 1.

Figura 5.24

Um caso de uso de referência.



livro=<título, autor, preço, número de páginas, editora, isbn, imagem de capa>
item=<título, autor, quantidade, preço unitário, subtotal>

Figura 5.25

A representação do fluxo principal de um caso de uso como um diagrama de sequência de sistema.

Note que, entre os atores e a interface, os fluxos consistem de envio e recebimento de informação. A informação pode ser enviada por um ator, por exemplo, a digitando em um campo ou pode ser recebida por um ator quando ela é impressa na tela, por exemplo. Neste nível, fluxos não são chamadas de métodos porque os atores não são objetos internos ao sistema.

As palavras ou expressões usadas nos fluxos do diagrama de sequência para representar informação algumas vezes são uma fonte de confusão ou entendimento falho. Para evitar tal confusão, um padrão de rotulação é recomendado:

- Informação simples (dados alfanuméricos) é representada por uma ou mais palavras. Por exemplo, *palavra-chave* (linha 1 da Figura 5.25), *quantidade* (linha 3) e *total* (linha 5).
 - Informação complexa pode ser representada entre “<” e “>” no diagrama se ela for composta de poucos elementos (2 ou 3). Por exemplo, <livro selecionado, quantidade> (passo 3).
 - Informação complexa composta de vários elementos (mais de 3) pode ser representada em uma nota separada ou dicionário de dados⁷. Apenas o nome do conceito complexo é usado no diagrama. Por exemplo, *livro* (passo 2) e *item* (passo 4).
 - Uma coleção de valores é indicada pelo sufixo “*”. Por exemplo, *palavra-chave** (passo 1), *livro** (passo 2), <livro selecionado, quantidade>* (passo 3) e *item** (passo 4).
 - Quando um objeto é selecionado de uma lista apresentada pelo sistema, a seleção é indicada pela palavra “selecionado”, como no passo 3, onde “livro selecionado” indica um livro que foi selecionado da lista *livro** apresentada no passo 2.
- Algumas outras considerações sobre o diagrama seguem:
- No passo 1, a equipe poderia escolher usar *palavras-chave* em vez de *palavra-chave**. Entretanto, neste caso, o fato de que a informação sendo enviada é uma lista de palavras não seguiria a notação proposta.
 - A informação contida em um *livro* é provavelmente um subconjunto dos atributos da classe *Livro* do modelo conceitual; ela não corresponde necessariamente a um conjunto completo de atributos da classe *Livro*.
 - No passo 3, quando um usuário informa ao sistema sobre a lista de livros que ele selecionou, não é necessariamente a informação completa sobre *livro* que é enviada pelo usuário para o sistema. Tecnicamente falando, o usuário precisa apenas selecionar um livro de uma lista e a interface vai enviar ao sistema o identificador correto para o livro selecionado (o usuário nem mesmo saberá que identificador é este).
 - No passo 4, um item foi criado para representar diferentes aspectos de um livro: em vez de *número de páginas*, *editora*, *ISBN* e *imagem de capa*), um item apresenta

⁷ Lembre-se de que nem sempre é o caso de que a informação complexa trocada com o sistema corresponde a um conjunto completo de atributos de um conceito do modelo conceitual.

quantidade (que não é um atributo de um livro) e *subtotal* (que é calculado a partir do preço e da quantidade).

- O passo 5 não representa informação alfanumérica sendo enviada ou recebida como nos passos anteriores. *Finalizar pedido* é um passo de controle. Ele apenas informa o sistema que o estado de um pedido mudou, sem passar qualquer outra nova informação alfanumérica.
- A Variante 5d é uma escolha entre finalizar o pedido ou continuar as compras. Ela é representada pelo fragmento *loop* que inclui os passos 1 a 4 do caso de uso.

5.8.3 Conexão da interface ao controlador-fachada

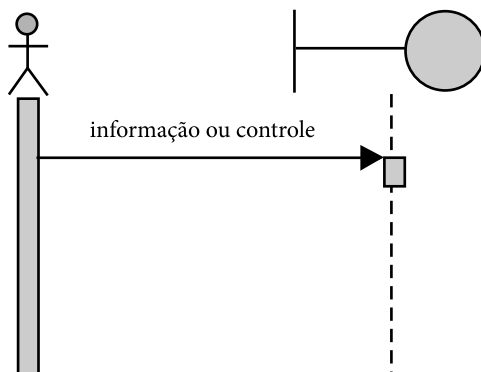
Eventos de sistema são ações que um usuário realiza sobre a interface de um sistema. Quando uma interface web é usada, por exemplo, essas ações consistem em preencher campos, pressionar botões etc. Elas não são operações no sentido de linguagens de programação, isto é, esses fluxos de um ator para o sistema e vice-versa não representam chamadas de métodos.

Entretanto, *comandos de sistema* e *consultas de sistema* são procedimentos computacionais que são necessários para implementar eventos de sistema e respostas de sistema, respectivamente. Comandos e consultas de sistema são ativados por mensagens que são enviadas pela interface para o controlador-fachada. Agora, temos um componente de sistema que invoca outro componente. Nesse caso, é a interface que envia uma mensagem, pedindo ao controlador-fachada que execute um método que consiste de um comando ou consulta. O conjunto de todas as possíveis mensagens é equivalente à funcionalidade completa do sistema: todos acessos e atualizações sobre a lógica dos dados são feitos por meio de comandos e consultas de sistema.

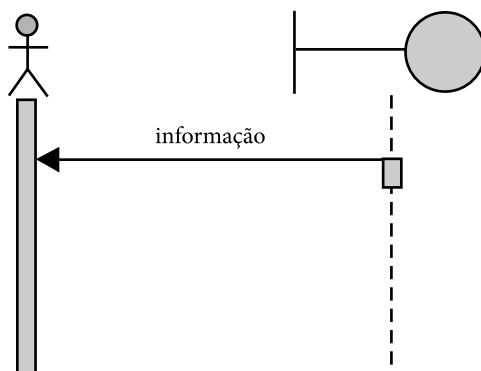
Assim, quatro tipos de fluxos são de interesse nos diagramas de sequência de sistema:

- *Evento de sistema*: como será mostrado na Figura 5.26, é uma ação realizada por um ator que envia alguma informação para o sistema. No diagrama, ele é representado por uma seta do ator para a interface.
- *Resposta de sistema*: como será mostrado na Figura 5.27, é um fluxo de informação do sistema para um dos atores, representado no diagrama como uma seta da interface para o ator.
- *Comando de sistema*: como será mostrado na Figura 5.28, é uma mensagem que é enviada da interface para o controlador, usualmente em resposta a um evento de sistema. O comando de sistema deve, por definição, modificar alguma informação armazenada ou gerenciada pelo sistema. No diagrama, ele é representado por uma seta da interface para o controlador rotulada com a mensagem (o uso de parênteses é sempre recomendado para distingui-la dos fluxos de informação como eventos e respostas de sistema).

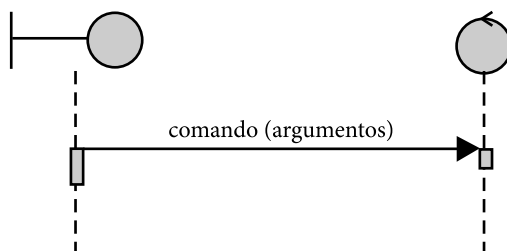
- *Consulta de sistema:* como mostrado na Figura 5.29, é uma mensagem enviada da interface para o controlador com o objetivo de obter alguma informação do sistema. Consultas não devem alterar dados; elas apenas retornam dados. No diagrama, consultas são representadas por setas da interface para o controlador rotuladas com uma mensagem com um valor de retorno explícito.

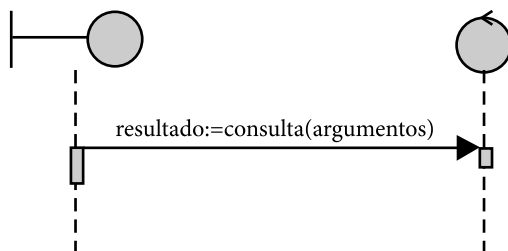
**Figura 5.26**

Evento de sistema.

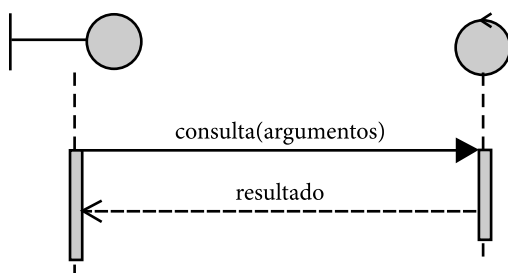
**Figura 5.27**

Resposta de sistema.

**Figura 5.28** Comando de sistema.


Figura 5.29

Consulta de sistema.


Figura 5.30

Representação alternativa para uma consulta de sistema.

Os resultados de uma consulta também poderiam ser representados como uma seta tracejada do controlador para a interface, como mostrado na Figura 5.30. Ambas representações são permitidas e a equipe pode escolher uma ou outra. A abordagem da Figura 5.29 cria menos setas no diagrama, mas pode ser complicada se dados complexos forem retornados pela consulta. O problema pode ser minimizado se notas como as da Figura 5.25 forem usadas para definir dados complexos. Outros analistas poderiam preferir a abordagem da Figura 5.30 que faz o retorno aparecer explicitamente como uma seta no diagrama. A desvantagem dessa abordagem é que ela divide uma única consulta em duas setas.

Comunicação entre atores pode ser representada no diagrama de sequência de sistema como setas de um ator para o outro. Entretanto, como explicado antes, essas comunicações não afetam a implementação do sistema.

A camada de domínio de uma aplicação (a parte do sistema que contém todas as classes que realizam as operações lógicas sobre os dados) é encapsulada por seu controlador-fachada, o qual é uma instância de uma classe que implementa todos os comandos e consultas de sistema que devem ser acessados por uma dada interface ou conjunto de interfaces. Existem pelo menos três subpadrões para o controlador-fachada. O primeiro, para sistemas menores, consiste em implementar um único controlador para todo o sistema. Entretanto, isso poderia resultar em um grande número de operações sendo implementadas

em uma única classe. Uma solução para isso é dividir a classe em *controladores de caso de uso*, ou seja, uma classe controladora diferente para cada caso de uso. Entretanto, esta não é uma boa solução quando diferentes casos de uso chamam os mesmos métodos, que então deverão ser implementados em diferentes classes. A solução mais adequada para a maioria dos sistemas de médio e grande porte é implementar *controladores de componente* ou de *subsistema*, ou seja, controladores que encapsulam a funcionalidade de subsistemas ou componentes conforme definido pela arquitetura do sistema.

O design das mensagens entre a interface e o controlador é feito após um exame dos eventos e respostas de sistema usando as seguintes regras:

- Um evento de sistema que envia para o sistema dados que serão armazenados, atualizados ou removidos, ou seja, dados que causam uma mudança no estado interno do sistema requer pelo menos um comando de sistema (Figura 5.31).
- Uma resposta de sistema que envia dados do sistema para o usuário requer pelo menos uma consulta de sistema, de forma que os dados possam ser obtidos do controlador para serem apresentados pela interface (Figura 5.32).
- Uma sequência de evento e resposta de sistema, onde o evento apenas fornece argumentos para produzir a resposta, requer uma consulta na qual os argumentos consistem na informação recebida pelo evento de sistema (Figura 5.33).

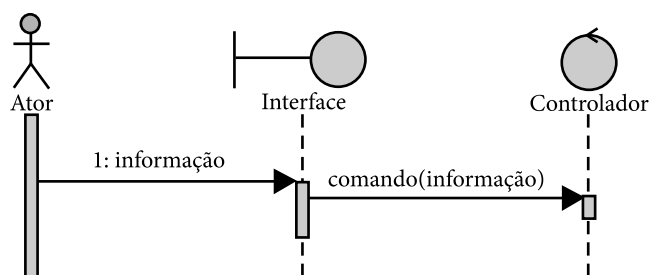


Figura 5.31

Um evento de sistema que requer um comando de sistema.

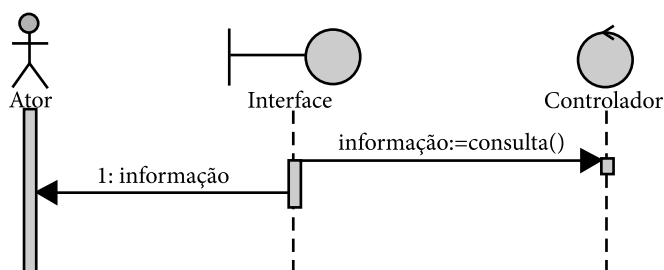
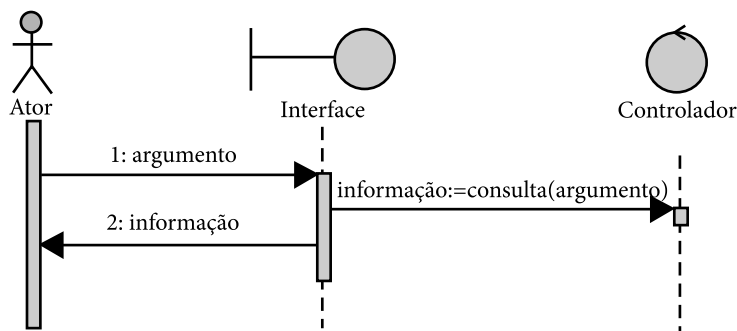


Figura 5.32

Uma resposta de sistema que requer uma consulta de sistema.


Figura 5.33

Uma sequência de evento e resposta de sistema que requer uma consulta de sistema, pois o evento apenas repassa argumentos para a consulta.

A relação de causa e consequência entre eventos e comandos e entre respostas e consultas não é sempre tão direta. Algumas vezes, uma consulta pode usar argumentos que já foram passados para a interface vários passos antes; ou informação passada pelo ator para a interface por ser usada muitas vezes por diferentes comandos e consultas.

Essas regras para derivar consultas e comandos das respostas e eventos são apenas uma primeira aproximação do design da camada de interface. Mais tarde, técnicas de modelagem e decisões de design podem mudar a forma como esses comandos e consultas são invocados.

5.8.4 Estratégia *stateless*

Quando um diagrama de sequência de sistema está sendo desenvolvido, cada peça de informação é passada do ator para a interface uma única vez. Entretanto, no nível seguinte, entre a interface e o controlador, vários comandos e consultas diferentes podem necessitar dos mesmos argumentos. Por exemplo, o CPF do comprador pode ser necessário para realizar várias operações subsequentes. Neste ponto, o designer deve decidir se o controlador deve ter memória temporária para armazenar esses argumentos (*estratégia stateful*) ou se ele não é provido com este tipo de memória (*estratégia stateless*). Na *estratégia stateless*, cada vez que uma consulta ou comando de sistema precisa de um argumento, ele deve ser recebido explicitamente da interface.

A Figura 5.34 mostra como o diagrama de sequência de sistema poderia se parecer para o caso de uso *Pedir livros* usando a *estratégia stateless*. A informação é passada dos atores para a interface apenas uma vez, mas cada vez que um comando ou consulta de sistema precisa dessa informação, a interface deve enviá-la de novo como um argumento para o controlador. Observe que *idCarrinho* é passado pela interface para o controlador três vezes: quando *adicionarNoCarrinho*, *consultarSumárioDoCarrinho* e *finalizarPedido* são chamadas.

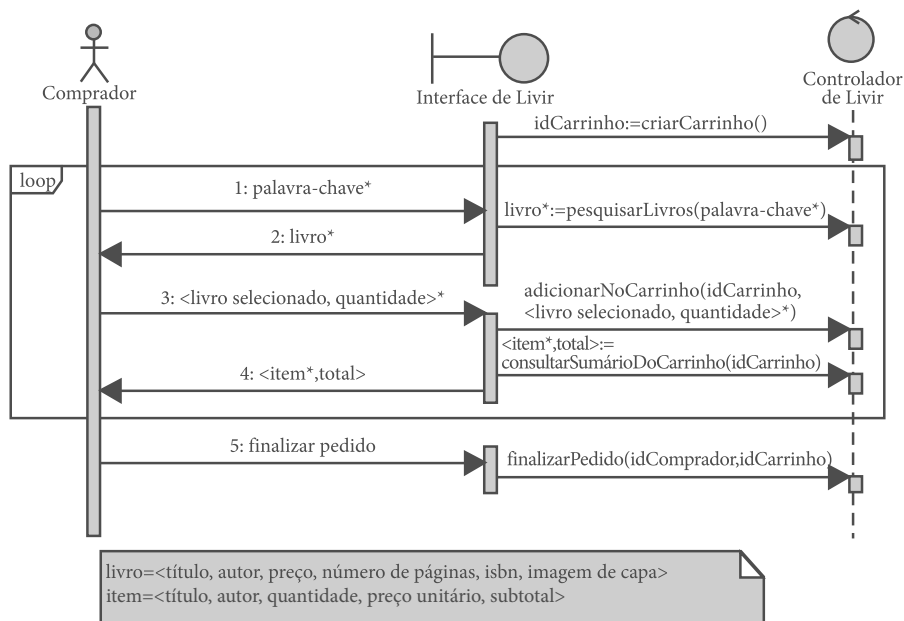
**Figura 5.34**

Diagrama de sequência de sistema para o fluxo principal do caso de uso *Pedir livros* usando a estratégia *stateless*.

A partir desse ponto, percebe-se que a equipe está fazendo uma transição da análise para o design, porque decisões de design precisam ser feitas para acomodar os requisitos. Por exemplo, o fato de que um usuário deve ser identificado apenas quando o pedido é finalizado evita que o sistema crie um pedido antes daquele ponto (se o pedido é realmente um conceito que precisa ser associado a um comprador). Portanto, um carrinho de compras é criado em vez do pedido na inicialização do diagrama de sequência. O pedido será criado apenas quando o usuário decidir parar de comprar (finalizar o pedido). Neste ponto, se o usuário ainda não se identificou, uma exceção poderia ser levantada, como explicado adiante. Um ponto que deve ser lembrado é que o controlador não tem memória temporária: ele não registra a informação de quem está comprando. É a interface que mantém a identificação do comprador neste momento. Quando um pedido está pronto para ser completado, a interface envia ao controlador os identificadores do comprador e do carrinho de compras (lembre-se de que o carrinho de compras ainda não está associado a nenhum comprador).

Como o controlador não tem estado (memória temporária), cada mensagem enviada para ele que se refira ao carrinho de compras deve enviar o identificador do carrinho (*id-Carrinho*). Por que se usa aqui o identificador do carrinho em vez do próprio objeto? Não estamos fazendo design orientado a objetos? O ponto é que os objetos de domínio estão encapsulados pelo controlador. Isso significa que a interface não pode ter acesso direto

aos objetos de domínio. É por isso que a interface envia um identificador alfanumérico como argumento em vez do objeto *carrinhoDeCompras*.

Um comentário deve ser feito sobre o comando *criarCarrinho*: como ele cria um novo objeto de domínio e retorna seu código identificador, alguém poderia perguntar se isso não fere o princípio de separação comando-consulta. Isso não acontece. Esta é uma das exceções aceitáveis ao princípio porque o comando simplesmente cria um novo objeto e retorna seu código de identificação. Isso pode ser feito porque tentar obter o código de um objeto recém-criado por outros meios poderia ser contraproduutivo.

5.8.5 Estratégia *stateful*

Quando o controlador pode ter memória temporária para guardar parâmetros de trabalho durante a execução de um caso de uso, então estaremos usando a estratégia *stateful*. Assim, após um argumento ser enviado ao controlador por uma operação, as operações subsequentes não precisariam mais receber este argumento; o controlador poderia manter esse valor em memória local.

Entretanto, o argumento enviado não é necessariamente uma informação persistente; a informação temporária não é armazenada no banco de dados ou outra estrutura equivalente. Ela consiste apenas de informação temporária (por exemplo, quem é o comprador corrente, qual é o carrinho de compras sendo usado etc.), a qual está disponível apenas durante a transação do caso de uso. A Figura 5.35 apresenta o mesmo caso de uso que foi mostrado na Figura 5.34, mas, dessa vez, usando a abordagem *stateful*. Pode-se observar que, entre outras coisas, não é necessário retornar o código identificador do novo carrinho de compras, porque o controlador vai lembrar-se dele.

Com a estratégia *stateful*, não é necessário retornar a referência ao novo carrinho de compras porque o controlador tem um mecanismo interno para guardar esse valor. Da mesma forma, os métodos *adicionarNoCarrinho*, *consultarSumarioDoCarrinho* e *finalizarPedido* também não precisam receber esse argumento. Ele é mantido pelo controlador. Da mesma forma, a identificação do comprador, se conhecida, é mantida pelo controlador, não pela interface.

Uma opção para implementar essa memória temporária é o uso de associações temporárias, como explicado na Seção 8.3. É uma decisão arquitetural escolher entre uma ou outra abordagem. Alguns prós e contras são os seguintes:

- A estratégia *stateful* requer a implementação de um mecanismo de memória temporária (não persistente) para armazenar alguns parâmetros (o uso de associações temporárias, por exemplo). A estratégia *stateless* não requer esse tipo de mecanismo.
- A estratégia *stateless* requer mais passagem de parâmetros entre a interface e o controlador. Quando for o caso de passar informação por meio de uma rede, isso pode se tornar inconveniente. Com a estratégia *stateful*, cada informação é transmitida uma única vez.

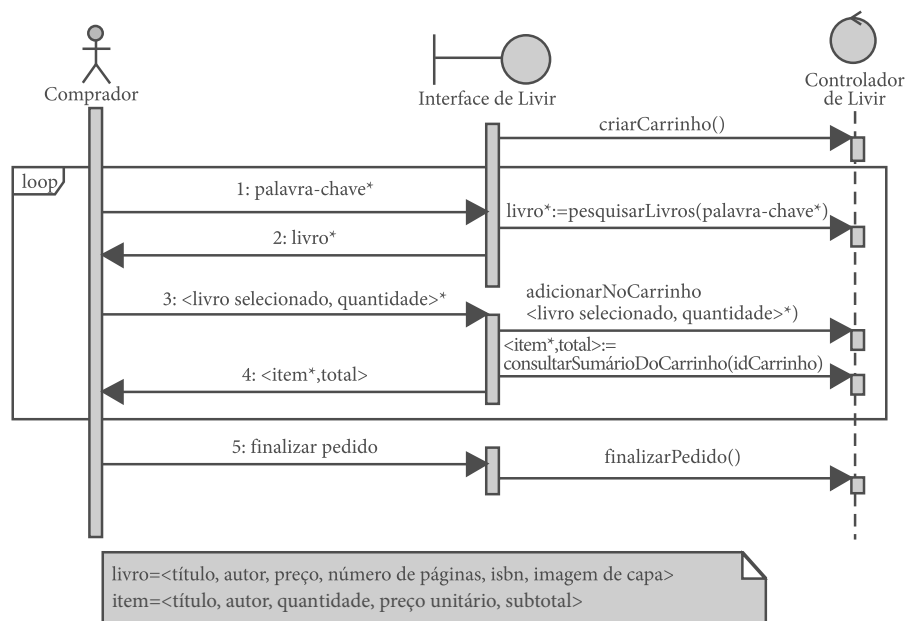
**Figura 5.35**

Diagrama de sequência de sistema para o fluxo principal do caso de uso *Pedir livros* usando a estratégia *stateful*.

Adicionalmente, pode-se dizer que as operações se tornam mais reusáveis quando se usa a estratégia *stateless*, porque cada uma delas recebe todos os parâmetros que precisa sem depender de outras operações que tenham sido executadas antes. Além disso, a estratégia *stateless* permite lidar com o crescimento do sistema melhor do que a estratégia *stateful*. Sistemas com muitos usuários concorrentes seriam mais fáceis de produzir e manter com a estratégia *stateless*.

5.8.6 Fluxos alternativos em diagramas de sequência de sistema

Como visto no início deste capítulo, passos de casos de uso, especialmente eventos de sistema, podem apresentar exceções, as quais são tratadas por um fluxo alternativo.

A inserção desses fluxos alternativos em um diagrama de sequência de sistema pode ser feita em etapas. Inicialmente, as exceções podem ser indicadas no diagrama bem como o local onde elas ocorrem. Então um fragmento *opt* pode ser produzido, ou seja, uma parte opcional do diagrama de sequência, como será mostrado na Figura 5.36. O procedimento consiste em tomar o caso de uso completo, como o que foi mostrado na Figura 5.7 e adicionar para cada exceção um fragmento *opt* na posição adequada do diagrama (usualmente logo após o passo que poderia causar a exceção).

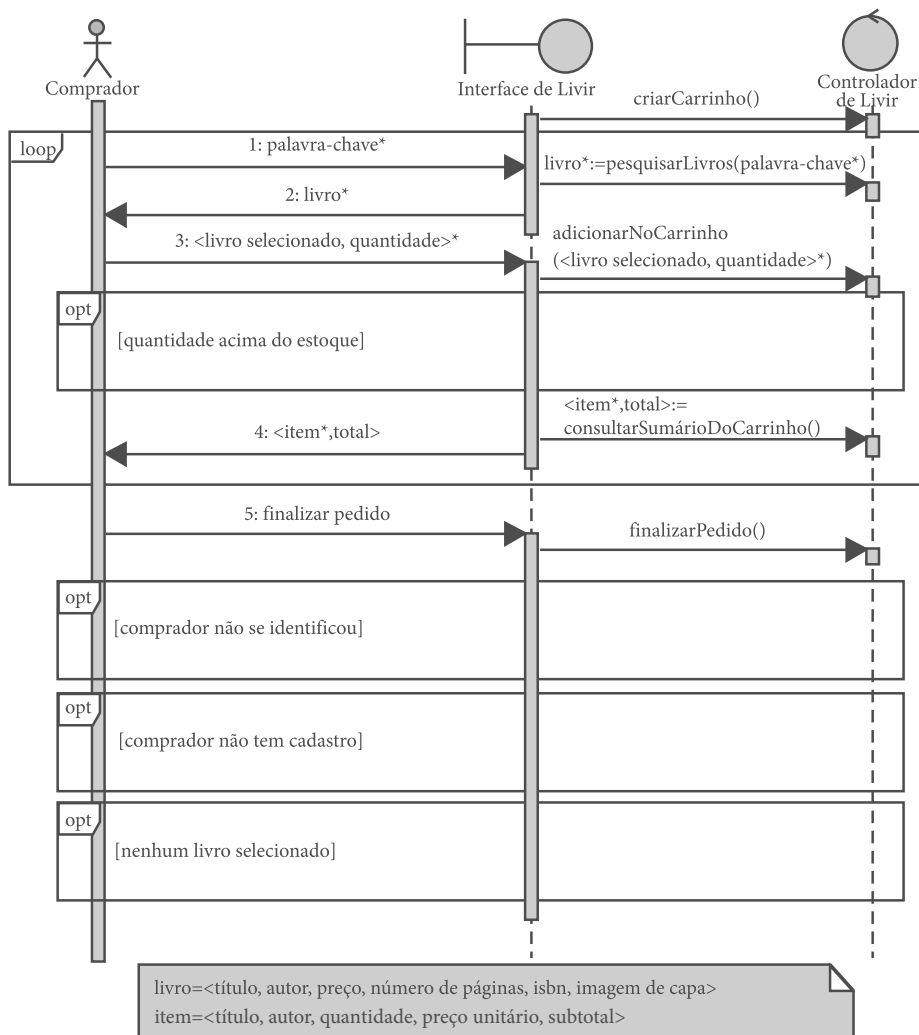


Figura 5.36

Diagrama de sequência de sistema com exceções incluídas.

O diagrama da Figura 5.36 mostra literalmente que, após os passos 3 e 5, eventos opcionais podem ocorrer. Esses eventos correspondem às exceções 3a, 5a, 5b e 5c que foram mostradas na Figura 5.7.

Agora, é necessário projetar o que acontece dentro dos fragmentos opcionais quando cada exceção é tratada. O fragmento *opt* contém os passos do fluxo alternativo, mas também é importante olhar com cuidado a forma como este fluxo termina. Alguns dos fluxos alternativos retornam para o passo que gerou a exceção e devem ser tratados como laços repetitivos que incluem o passo que causou a exceção. Outros avançam para o próximo passo e deveriam ser tratados simplesmente como um fragmento

opcional do diagrama. Quando outros casos de uso são referenciados ou quando um fragmento é tão longo que justifica sua definição em um diagrama separado, então o fragmento *ref* pode ser usado, indicando que outro diagrama de sequência está sendo invocado. O fragmento *ref* pode receber e retornar informação da mesma forma que uma função de programação o faz, e ele usa uma notação similar, como, por exemplo, *ref outroDiagrama(argumentos):resultado*.

O diagrama da Figura 5.36 será evoluído com o tratamento de exceções. Isso pode se tornar uma tarefa complexa, a não ser que algumas decisões de design sejam tomadas com cuidado. Por exemplo, no passo 3, um conjunto de livros e respectivas quantidades é enviado para o sistema e apenas então as quantidades são verificadas. Isso poderia requerer tratar uma exceção na qual alguns livros são pedidos acima do estoque e outros não, o que poderia ser inconveniente. Portanto, foi decidido modificar o diagrama de sequência de forma que a exceção será verificada cada vez que uma quantidade for informada. Note que esta decisão muda um pouco a forma como o usuário percebe o funcionamento do sistema: em vez de reclamar da quantidade acima do estoque apenas no final da seleção de vários livros, o sistema vai verificar essa quantidade após cada seleção de um livro. O diagrama resultante é mostrado na Figura 5.37, em que apenas os três primeiros passos do caso de uso são representados por simplicidade.

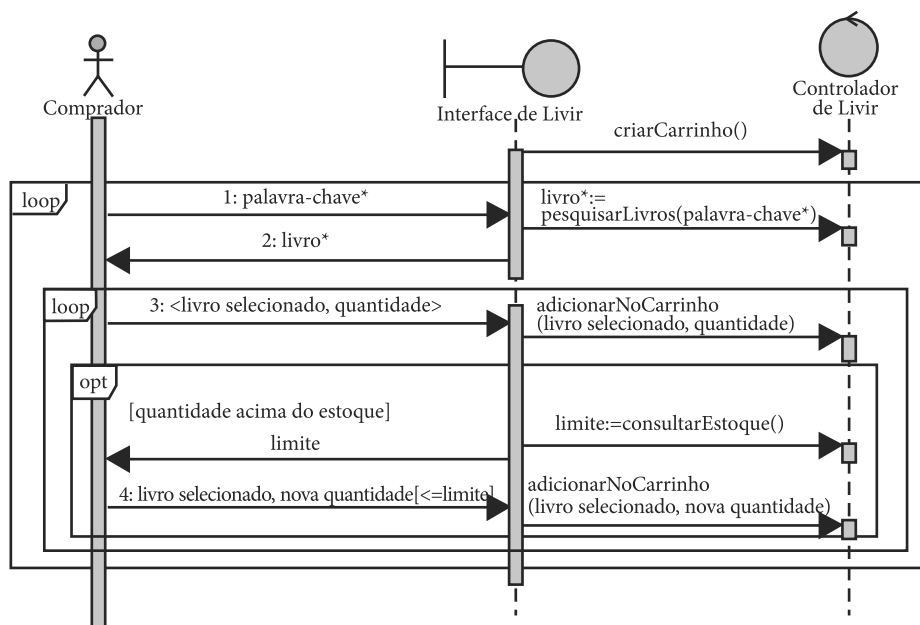
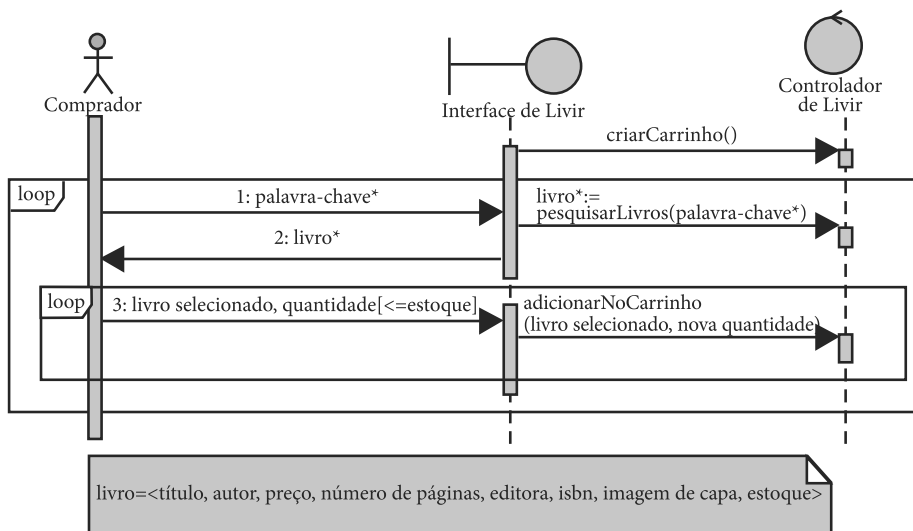


Figura 5.37

Parte de um diagrama de sequência de sistema com uma mudança de design e um tratador de exceção.

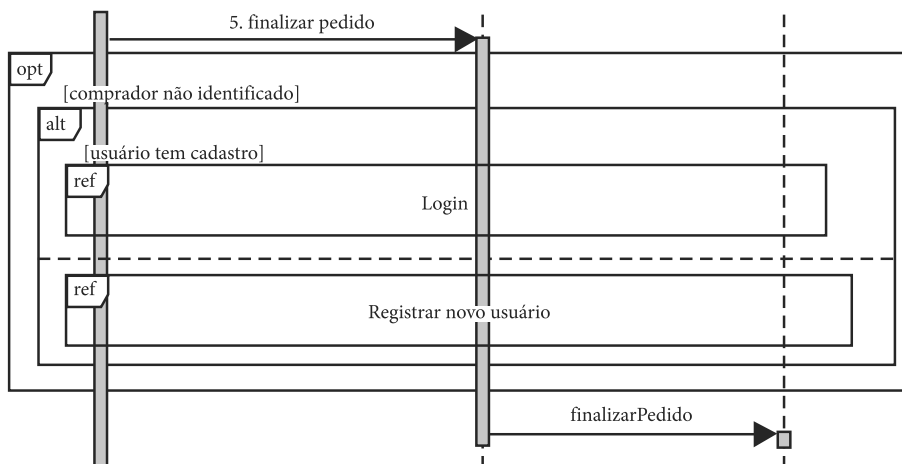

Figura 5.38

Parte de um diagrama de sequência de sistema no qual um tratador de exceção foi reprojeto.

Note que na Figura 5.37 o design poderia ser melhorado novamente. Por que tentar realizar um comando com uma quantidade acima do estoque se o sistema já conhece essa informação? A informação sobre o estoque poderia ser retornada pela consulta *pesquisarLivros* de forma que o campo de entrada na interface poderia já ser limitado pela quantidade real disponível. Esta informação poderia também ser usada pela interface para mostrar os livros que estão em catálogo, mas não no estoque como *em falta*. A Figura 5.38 mostra o design resultante.

Observe que na Figura 5.38 o fragmento que trata a exceção simplesmente desaparece. Isso acontece porque a exceção foi transformada em précondição para o comando de sistema *adicionarNoCarrinho*. Esta é uma técnica popular para melhorar o design que será revista no Capítulo 8. A técnica consiste em evitar que o usuário envie um argumento inválido; a escolha do usuário é limitada a valores válidos. Assim, a exceção não pode mais acontecer.

As exceções 5a e 5b podem ser resolvidas juntas, já que elas envolvem a identificação do usuário. Quando o comando *finalizarPedido* é executado, uma exceção poderia ocorrer se o usuário ainda não estivesse identificado. O usuário teria então duas escolhas: entrar com uma identificação válida ou criar uma nova. O diagrama parcial é mostrado na Figura 5.39.

**Figura 5.39**

Lidando com a identificação do usuário.

Como mostrado na Figura 5.39, não é necessário tentar o comando *finalizar-Pedido* para descobrir que o usuário ainda não foi identificado. A condição pode ser testada depois que o usuário decidir finalizar o pedido, mas antes de tentar executar o comando. Assim, se o usuário ainda não foi identificado, ele tem duas opções, representadas dentro do fragmento *alt*. O fragmento *alt* funciona como um “se-então-senão”: se o usuário tem uma conta, então ele deve fazer seu login, senão ele deve criar uma nova conta. Em ambos os casos, como essas sequências podem ser altamente reusáveis, uma referência a outro diagrama de sequência é indicada pelo fragmento *ref*.

Finalmente, a exceção 5c estabelece que, se nenhum livro foi selecionado, o pedido não pode ser fechado. Isso pode ser implementado como um laço repetitivo (*loop*) colocado sobre os fluxos 1 a 4 (excluindo o fluxo 5) que iria repetir até que pelo menos um livro fosse selecionado. Como já existe este laço no diagrama em razão da variante 5d (veja a Figura 5.37), então basta adicionar a condição ao laço existente. O diagrama de sequência completo com todas as exceções tratadas é mostra a Figura 5.40, a seguir.

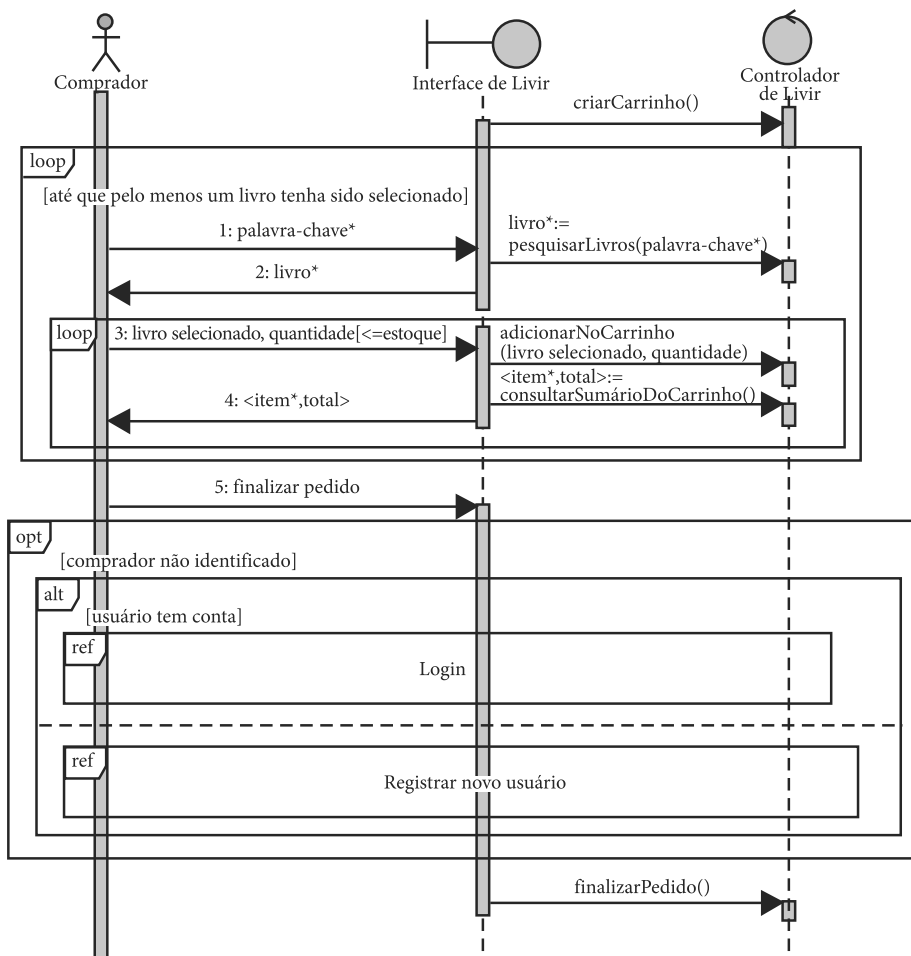


Figura 5.40

Diagrama de sequência completo com exceções tratadas.

O principal resultado prático do diagrama de sequência de sistema é a descoberta e design dos comandos e consultas de sistema que devem ser implementadas para permitir que a funcionalidade do sistema esteja disponível para os usuários. De acordo com a Figura 5.40, esses comandos e consultas são os seguintes⁸:

- *criarCarrinho()*
- *pesquisarLivro(palavra-chave*):livro**
- *adicionarNoCarrinho(livro selecionado, quantidade)*
- *consultarSumárioDoCarrinho():<item*, total>*
- *finalizarPedido()*

⁸ A estratégia aplicada na Figura 5.40 é *stateful*. Se a estratégia *stateless* fosse usada em vez dela, as operações seriam ligeiramente diferentes, porque elas teriam mais parâmetros.

O Capítulo 8 mostra como usar esta informação para construir um conjunto de contratos que definem a funcionalidade completa do sistema.

5.9 O processo visto aqui

	Concepção	Elaboração
Modelagem de negócio		
Requisitos		Detalhar os requisitos, expandindo os casos de uso: • Identificar o fluxo principal. • Identificar os fluxos alternativos: variantes e tratadores de exceção
Análise e design		Elaborar o diagrama de sequência de sistema: • Representar o fluxo principal de um caso de uso como um diagrama de sequência de sistema. • Representar os comandos e as consultas de sistema, usando a estratégia <i>stateful</i> ou <i>stateless</i>. • Completar o diagrama de sequência de sistema com fluxos alternativos.
Implementação		
Teste		
Gerenciamento de Projeto		

5.10 Questões

1. Variantes e tratadores de exceção são fluxos alternativos para um caso de uso. Em quais situações um ou outro deveria ser usado?
2. Qual a diferença entre um caso de uso essencial e um caso de uso real? Por que o caso de uso essencial é mais interessante para a análise de requisitos?
3. Explique as diferenças entre passos de caso de uso obrigatórios, complementares e impróprios.
4. Expanda o fluxo principal do caso de uso 03, *Enviar pedido*, apresentado na Figura 3.10.
5. Finalize a expansão do caso de uso da Figura 5.2 indicando seus fluxos alternativos.
6. Projete o diagrama de sequência de sistema para o fluxo principal do caso de uso da Figura 5.2, escolhendo uma das seguintes estratégias: *stateful* ou *stateless*. Após projetar o diagrama para o fluxo principal, adicione os fluxos alternativos identificados na Questão 5.

Modelagem conceitual: fundamentos

TÓPICOS-CHAVE NESTE CAPÍTULO

- Atributos
- Conceitos
- Associações
- Coleções
- Organização do modelo conceitual: estrutural, associativa e temporal
- Invariantes
- Construção iterativa do modelo conceitual

6.1 Introdução à modelagem conceitual

A análise de domínio consiste em descobrir e modelar a informação que deve ser gerenciada pelo sistema. Isso significa que a equipe deve descobrir como a informação deve ser estruturada e transformada. A análise de domínio inicia durante a Concepção com o modelo conceitual preliminar e continua durante a Elaboração, quando este modelo é refinado e completado à medida que a análise de requisitos é aprofundada com a expansão dos casos de uso.

Pode-se analisar dois aspectos da informação: o *estático* (também chamado de *estrutural*), que é estudado neste capítulo e no Capítulo 7, e o *funcional*, estudado no Capítulo 8. O aspecto estático pode ser representado no modelo conceitual e o aspecto funcional nos contratos de operação de sistema.

Durante a análise, não há *modelo dinâmico* para os objetos, porque a análise toma em consideração apenas uma visão externa do sistema. O modelo dinâmico, consistindo das colaborações entre os objetos, é desenvolvido durante o design (Capítulo 9), porque apenas naquele momento os aspectos internos do sistema serão tratados. Assim, o modelo funcional da análise indica apenas quais informações devem entrar e sair do sistema, sem dizer como a informação é transformada. Mais tarde, o modelo dinâmico do design vai mostrar claramente como as colaborações entre objetos poderiam transformar a informação.

O modelo conceitual descreve a informação que o sistema vai gerenciar. Ele é um artefato do domínio do *problema*, não do domínio da *solução*. Portanto, o modelo conceitual não deve ser confundido com o *Diagrama de Classes de Design* (DCD), o qual pertence à

arquitetura do software (Capítulo 9). O DCD, embora inicialmente derivado do modelo conceitual, pertence ao domínio da solução e, portanto, serve a um propósito diferente.

O modelo conceitual também não deve ser confundido com o *modelo de dados* (Capítulo 13), porque este enfatiza a representação e a organização dos dados armazenados, enquanto o modelo conceitual tem como objetivo representar a compreensão da informação por parte dos usuários, e não sua representação física. Assim, o modelo de dados relacional é apenas uma das possíveis representações físicas de um modelo conceitual mais essencial.

Uma maneira interessante de compreender o modelo conceitual é imaginar que os elementos descritos por ele correspondem a informações que inicialmente existem apenas na mente do usuário, como representado na Figura 6.1, e não em um sistema de armazenamento físico.

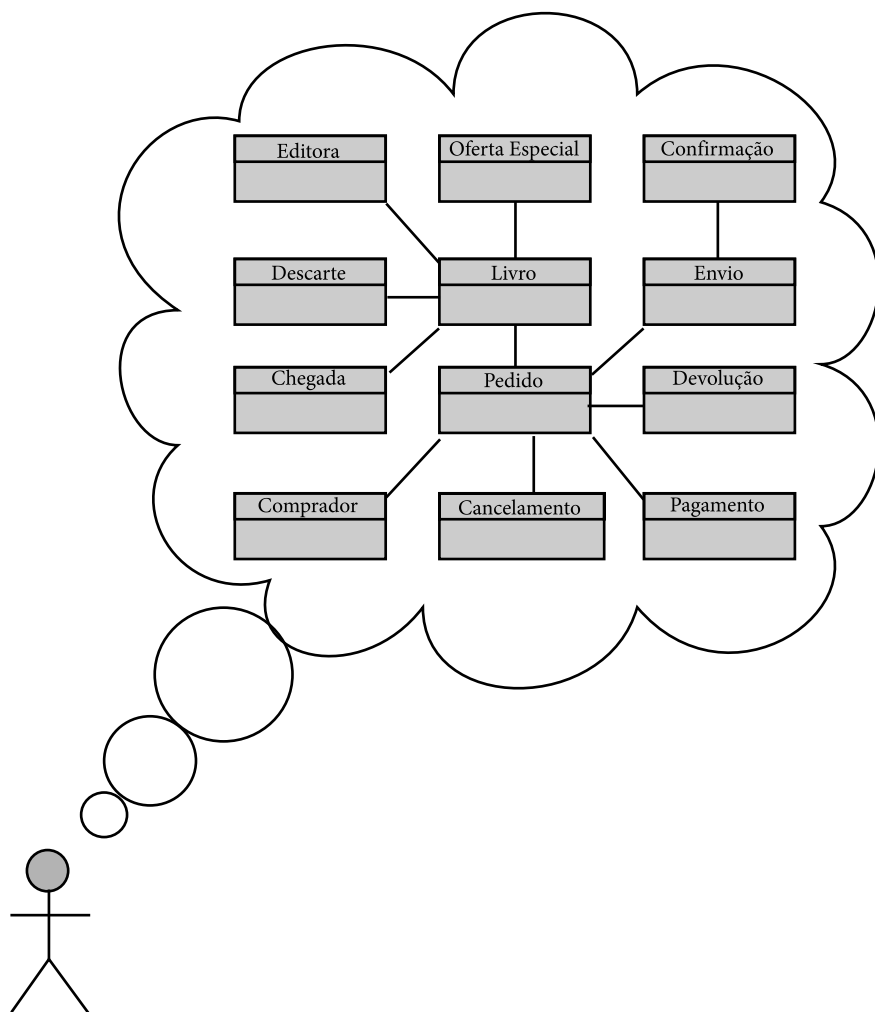


Figura 6.1

O modelo conceitual é uma representação da visão do usuário sobre a informação.

Um usuário envia e recebe informações do sistema por meio de eventos e respostas de sistema, respectivamente. Neste ponto, o sistema ainda não precisa ser considerado nem sequer como um sistema *computacional*, porque a informação existe independentemente do suporte computacional para armazená-la e gerenciá-la.

O objetivo da análise é estudar o problema. Mas um sistema computacional é a solução, e, portanto, pertence ao design. O sistema-solução poderia, inclusive, ser projetado sem tecnologia computacional. É possível analisar uma situação inteira e então propor uma solução manual para implementá-la, na qual, por exemplo, os dados são armazenados em fichas de papel e as operações são realizadas por trabalhadores usando lápis, borracha e grampeadores.

Assim como casos de uso essenciais, o modelo conceitual é independente de soluções físicas que poderiam ser adotadas e deveria conter apenas elementos que pertencem ao domínio do problema, deixando para o design os elementos da solução, ou seja, todos os elementos relacionados à tecnologia, tais como interfaces, armazenamento, comunicação, e assim por diante.

O modelo conceitual representa apenas os aspectos estáticos da informação. Portanto, no modelo conceitual, referências a operações ou aspectos dinâmicos do sistema não podem existir. Embora o modelo conceitual seja representado pelo diagrama de classes da UML, o analista *não deve ainda incluir quaisquer métodos*. O Capítulo 9 mostra como identificar e projetar métodos usando uma abordagem sistemática.

Quando um diagrama de classes é usado para modelagem conceitual, existem precisamente três tipos de elementos a serem usados para representar informação:

- *Atributos*: informação simples alfanumérica ou primitiva tal como números, textos, datas etc. Exemplos de atributos do projeto Livir são o nome do comprador, a data de pagamento, o título do livro e o valor total de um pedido. Um atributo está sempre conectado a um elemento mais complexo: o conceito.
- *Conceitos*: a representação de informação complexa que tem significado coerente no domínio. Conceitos usualmente agregam atributos e não podem ser descritos meramente como alfanuméricos. Conceitos também podem estar associados uns aos outros. Exemplos de conceitos no projeto Livir são livro, comprador, pedido e pagamento.
- *Associações*: um tipo de informação que liga diferentes conceitos. Entretanto, associações são mais do que meras ligações: elas *são* informação. No exemplo Livir, associações poderiam existir entre pedidos e comprador, e também entre pedidos e livros, por exemplo.

Estes três elementos são detalhados nas próximas seções. É praticamente impossível explicar um deles sem mencionar os outros porque os três são fortemente entrelaçados.

6.2 Atributos

Atributos são, no modelo conceitual, os elementos alfanuméricos e primitivos, tais como data, moeda, número, *string*, intervalo etc.

Embora a maioria das linguagens de programação permita que atributos sejam definidos como estruturas de dados tais como listas, *arrays*, conjuntos, árvores etc., isso não é recomendável na modelagem conceitual, porque essas estruturas são mais bem modeladas como associações, conforme será explicado nas seções seguintes.

Conceitos complexos (classes) também não deveriam ser usados como atributos de outros conceitos (embora, novamente, as linguagens de programação permitam isso). Por exemplo, um livro não deveria ser atributo de um pedido. Se um relacionamento entre livros e pedidos existe, então uma *associação* deveria ser usada em vez de atributo porque livros e pedidos são conceitos complexos. Uma exceção a esta recomendação ocorre quando um *tipo primitivo* (Seção 6.2.5) ou *enumeração* (Seção 6.2.4) é usado para rotular um atributo. Tipos primitivos e enumerações são definidos como classes estereotipadas, mas eles não se comportam como conceitos. Eles são usados como tipos de atributos. Por exemplo, uma data é composta de partes como ano, mês e dia, entretanto, a data de nascimento é considerada um atributo da pessoa e não um conceito complexo associado a ela.

Atributos em UML são sempre representados dentro de uma classe, como mostrado na Figura 6.2, onde a classe *Comprador* tem os atributos *nome*, *CPF*, *endereço* e *telefone*.

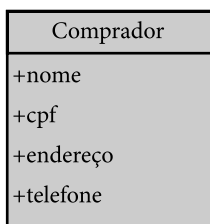


Figura 6.2

Atributos representados dentro de uma classe.

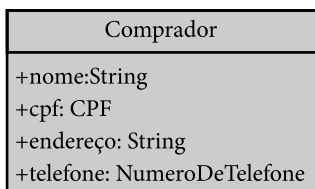


Figura 6.3

Atributos tipados.

6.2.1 Tipos de atributos

Atributos podem ter um tipo, embora isso não seja obrigatório para o modelo conceitual. A Figura 6.3 apresenta uma versão da classe da Figura 6.2 com atributos tipados.

Tipos em modelos conceituais têm o mesmo significado que em linguagens de programação. Na Figura 6.3, temos o tipo clássico *String*, e tipos primitivos definidos pelo usuário: *CPF* e *NumeroDeTelefone*.

Quando um atributo é definido por regras de formação, como no caso de *CPF* e *telefone*, é recomendável definir um tipo primitivo especialmente para ele, como foi feito na Figura 6.3.

O *endereço* é algo especial. Ele seria um atributo ou conceito complexo? Ele seria simplesmente uma *string* ou um conceito complexo composto de rua, número, CEP, município etc.? Este caso, como muitos outros, é decidido pela análise das necessidades de informação dos usuários. Se endereços são usados apenas para impressão de envelopes e envio de correspondência, então eles podem se comportar como simples *strings* e ser representados por atributos tipados como *string*. Entretanto, se os endereços são usados para calcular distâncias e rotas, ou se eles são usados para agrupar compradores que vivem próximos uns dos outros, então eles se comportam como conceitos complexos e deveriam ser modelados como um tipo primitivo. Além disso, se eles têm uma estrutura de associações entre eles, então eles devem ser modelados como conceitos complexos.

6.2.2 Valores iniciais

Um atributo pode ser declarado com um valor inicial, ou seja, toda vez que uma instância do conceito for criada, aquele atributo automaticamente receberá um valor inicial, o qual pode mudar mais tarde se necessário.

No sistema *Livir*, um pedido pode ser criado, por exemplo, com um valor total que é inicialmente zero. Isso pode ser definido no diagrama de classes como mostrado na Figura 6.4.

A definição de um valor inicial para um atributo pode também ser feita com o uso da *Object Constraint Language (OCL)* (Object Management Group, 2010).

Toda expressão em OCL deve ser declarada em um *contexto* que corresponde a uma classe. A maioria das declarações OCL também tem um *subcontexto*, que consiste em uma propriedade da classe, ou seja, um atributo, papel de associação ou método. Quando da declaração do valor inicial para o atributo *valorTotal* na classe *Pedido*, o contexto é *Pedido* e o subcontexto é *valorTotal*. Assim, a expressão OCL inicia como:

Pedido
+data:Data +valorTotal:Moeda=0 +numero:Natural

Figura 6.4

Uma classe mostrando um atributo com valor inicial.

```
Context Pedido::valorTotal
```

Opcionalmente, o tipo do atributo pode ser adicionado:

```
Context Pedido::valorTotal:Moeda
```

Para indicar que a expressão OCL está definindo um valor inicial para um atributo, a cláusula *init*: é usada, seguida da expressão que, quando avaliada, produz o valor inicial para o atributo. Como no exemplo, o valor inicial é a constante 0, e a expressão pode ser definida como:

```
Context Pedido::valorTotal:Moeda
  init: 0
```

A expressão pode também ser simplificada pela omissão do tipo do atributo:

```
Context Pedido::valorTotal
  init: 0
```

Um atributo pode ter seu valor inicial definido por expressões mais complexas. Mais adiante, são apresentados exemplos de expressões complexas em OCL que podem ser usadas para inicializar atributos com valores produzidos por operações lógicas e matemáticas.

6.2.3 Atributos derivados

Atributos derivados são entendidos aqui apenas como valores alfanuméricos. Se um “atributo derivado” se refere a objetos ou estruturas de dados, então ele é, de fato, uma *associação derivada* (Seção 6.4.3). Atributos derivados diferem dos atributos normais porque eles não podem ser alterados diretamente. Em outras palavras, eles são *read only*¹.

Um atributo derivado é calculado e deve ser definido por uma expressão. No diagrama de classes, um atributo derivado pode ser representado por uma barra (/) precedendo o nome (e tipo) do atributo, seguindo-se um sinal de igual (=) e a expressão (OCL) que define o atributo. A Figura 6.5 mostra um exemplo no qual *lucro* é definido como a diferença entre *preço* e *custo*.

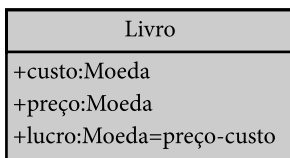
Atributos derivados podem ser também definidos por expressões OCL². A expressão novamente tem uma classe como contexto e um atributo como subcontexto. A isso se segue a palavra *derive*: que indica que a expressão que a segue define como o valor derivado é calculado. A expressão no exemplo da Figura 6.5 define *lucro* usando os valores de outros dois atributos da mesma classe: *lucro=preço-custo*.

Por ora, pode-se considerar que as expressões OCL após as cláusulas *init*: e *derive*: iniciam com uma expressão que denota uma instância da classe contexto. Essa instância é referenciada pela palavra *self*.

A notação *self.property* permite acessar o valor de uma propriedade (atributo, papel de associação ou método) de um objeto. Por exemplo, no contexto da classe *Livro*, a expressão *self.custo* denota o valor do atributo *custo* de uma dada instância de *Livro*.

¹ Apenas leitura.

² Não necessariamente, porque muitos analistas podem escolher a linguagem natural ou outra linguagem de especificação. Entretanto, como OCL é parte do pacote UML, é recomendável usá-la.


Figura 6.5

Uma classe mostrando um atributo derivado.

Assim, a expressão OCL que define o valor derivado do atributo *lucro* da classe *Livro* é:

```
Context Livro::lucro
derive:
    self.preço - self.custo
```

Em OCL, é possível omitir a expressão *self* quando o contexto for não ambíguo. No exemplo anterior, a expressão poderia ser simplificada para:

```
Context Livro::lucro
derive:
    preço - custo
```

No contexto definido, *preço* e *custo* não podem ser outra coisa senão atributos de *Livro*. Então, a palavra *self* pode ser omitida da expressão.

Atributos derivados não podem ser diretamente atualizados. Na classe *Livro* da Figura 6.5, apenas os atributos *custo* e *preço* podem ser diretamente atualizados. O atributo derivado *lucro* é *read only*: ele é calculado como o resultado da expressão que o define³.

6.2.4 Enumerações

Enumerações são um meio-termo entre conceitos e atributos. Elas são basicamente *strings*, e se comportam como tal. Mas há um conjunto predefinido de *strings válidas* que se constitui no domínio da enumeração. Por exemplo, um dia de semana só pode assumir uma dentre sete possibilidades: domingo, segunda-feira, terça-feira, quarta-feira, quinta-feira, sexta-feira ou sábado. Assim, um atributo definido como *DiaDaSemana* (tipado com essa enumeração) só poderia assumir um daqueles sete valores.

Enumerações podem aparecer nos diagramas da UML como classes estereotipadas. Sugere-se que as enumerações que são independentes de domínio não sejam colocadas no mesmo diagrama que contém o modelo conceitual, mas em um pacote criado especialmente para conter essas enumerações, como mostrado na Figura 6.6, já que enumerações independentes de domínio (tal como *DiaDaSemana*) são altamente reusáveis de

³ Neste ponto, alguns leitores que são também programadores podem ficar preocupados com questões relacionadas ao desempenho do código que será usado para implementar os atributos derivados. Como eles são por definição “calculados”, pode parecer uma perda de tempo de processador repetir estes cálculos, caso os valores originais de custo e preço não mudem. Entretanto, os leitores podem ficar calmos porque mecanismos de otimização de código estão disponíveis para atributos derivados. Por exemplo, o valor calculado para um atributo derivado pode ser mantido em *cache* e recalculado apenas quando um dos componentes mudar. Por exemplo, o *lucro* só precisa ser recalculado quando uma instância de *Livro* mudar os valores de *preço* ou *custo*.

uma aplicação para outra. Apenas enumerações dependentes de domínio deveriam ser deixadas junto às outras classes do modelo conceitual de forma que as pessoas pudessem ver seu significado mais facilmente.

Na Figura 6.6, o atributo *validade* da classe *Oferta* só pode ter atribuído um dentre os sete valores para a enumeração *DiaDaSemana*.

Enumerações podem ser consideradas classes estereotipadas. Embora elas até pudessem ter associações e atributos próprios, o analista não é encorajado a usar estas características porque elas podem transformar a enumeração em um conceito complexo que provavelmente perderia potencial de reusabilidade. Por exemplo, considere a enumeração *DiaDaSemana*. Ela é simplesmente uma lista de sete nomes. Suponha agora que o analista adicione alguns atributos à enumeração *DiaDaSemana*, tais como, por exemplo, as horas de trabalho. Então, por exemplo, de segunda à sexta, teriam horas de trabalho das 8:00 às 12:00 e das 14:00 às 18:00, sábado das 8:00 às 12:00 e domingo sem horas de trabalho. Diferentes aplicações poderiam definir as horas de trabalho de formas distintas. Neste caso, qual a diferença entre a enumeração e um conceito complexo, já que ela tem atributos, possibilidade de definir associações ou mesmo ter métodos, mais adiante? Em vez de criar atributos ou associações para uma enumeração, seria mais recomendável criar um novo conceito complexo que acomodasse a enumeração como um de seus atributos. No exemplo, um novo conceito complexo chamado *DiaDeTrabalho* poderia ser criado com dois atributos: *diaDaSemana* (mantido como enumeração) e *horasDeTrabalho* (um atributo normal, como explicado acima). Assim, enumerações que deveriam incorporar funcionalidades extras podem ser mantidas como são sem a estrutura usualmente incluída em classes normais.

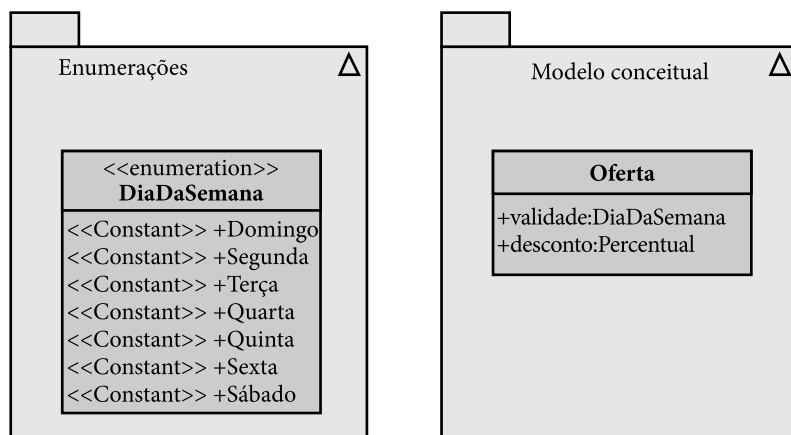
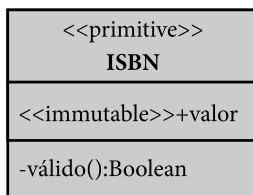


Figura 6.6

Definição e uso de uma enumeração.


Figura 6.7

Exemplo de um tipo primitivo.

Para permanecer reusável, uma enumeração não deveria ser capaz de acessar nenhuma outra classe por meio de uma associação, embora outras classes possam acessar a enumeração. O fato de que a *Oferta* na Figura 6.6 tenha um atributo rotulado como *DiaDaSemana* permite que a classe normal *Oferta* acesse a enumeração *DiaDaSemana*, mas não vice-versa. Isso mantém a enumeração desacoplada das classes que a usam e, portanto, ela permanece reusável.

Valores de enumeração em OCL podem ser representados pelo nome da enumeração seguido de “::” e o nome do valor, como, por exemplo, *DiaDaSemana::terça*.

6.2.5 Tipos primitivos

A equipe pode e deve definir *tipos primitivos* quando estiver lidando com atributos que tenham regra de formação, como no caso do ISBN. Tipos primitivos podem ser definidos como classes estereotipadas com <<primitive>>, como mostrado na Figura 6.7 a seguir.

A classe *ISBN* estereotipada desta forma não é um conceito complexo como *Livro* ou *Comprador*, mas um *tipo* que pode ser usado para definir atributos. Ela tem um atributo *valor* com acesso público (marcado com “+”) e um predicado de validação que é privativo (marcado com “-”), significando que ele só pode ser acessado por métodos internos à classe. Este predicado pode ser chamado para verificar se o valor do ISBN é válido ou não, considerando a regra de formação⁴. Esse predicado pode ser usado pelo *constructor* (construtor, o método que cria instâncias de uma classe) para garantir que apenas ISBNs válidos sejam criados. Lembre-se de que as classes conceituais, por enquanto, não têm métodos; mas tipos primitivos não são parte do modelo conceitual. Eles são componentes de baixo nível reusáveis definidos pela equipe.

Embora o valor do tipo primitivo tenha acesso público, é normalmente aceito que ele seja imutável. Isso significa que o valor do tipo primitivo só pode ser definido em tempo de criação da instância, e não pode mais ser atualizado depois disso. Isso é assim para deixá-lo consistente com a ideia de elementos de dados primitivos como *constantes*, e não como *objetos*. Objetos podem mudar seu estado interno, mas constantes usualmente não⁵.

⁴ A regra de formação do ISBN é explicada em <www.isbn.org/standards/home/isbn/us/isbnqua.asp>.

⁵ Algumas linguagens, como Smalltalk, permitem que constantes mudem seu valor em tempo de execução, mas isso normalmente não é recomendado para a maioria das aplicações.

Alguns tipos primitivos, tais como *Data* ou *Moeda*, já estão disponíveis em algumas linguagens de programação e sistemas de bancos de dados. Outros tipos não tão comuns em linguagens tais como ISBN, CEP, números ímpares etc. podem ser criados.

A regra de formação do tipo primitivo deve ser definida sintaticamente, isto é, pela avaliação de expressões que não necessitam consultar os dados gerenciados pelo sistema. Se a regra de formação depender de consultar atributos ou valores de associações, então provavelmente não se trata de um tipo primitivo, mas de uma classe normal. Por exemplo, a regra de formação de um *número primo* pode ser verificada sem a necessidade de observar quaisquer dados armazenados, mas um *nome de comprador cadastrado* não poderia ser um tipo primitivo, porque, para verificar se uma *string* corresponde a um nome de comprador existente, é necessário consultar a lista de compradores que é gerenciada pelo sistema.

Um tipo primitivo usualmente é *independente de aplicação*, isto é, seu significado usualmente é independente do sistema ou projeto sendo desenvolvido. Um ISBN tem o mesmo significado e regra de formação em qualquer aplicação que faça uso dele. Já conceitos como *Comprador*, *Livro* ou *Pedido* podem ter definições diferentes em aplicações distintas e usualmente não podem ser reusados sem algum esforço de refatoração. Entretanto, tipos primitivos usualmente são 100% reusáveis, sem ajustes nem modificações.

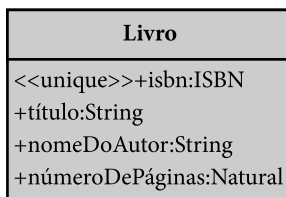
6.3 Conceitos

Conceitos são mais do que valores alfanuméricos. Eles são também mais do que um agregado de atributos porque eles carregam significado e podem ser associados uns com os outros. A modelagem conceitual pode começar com conceitos e associações apenas, como mostrado na Figura 3.6, ou seja, atributos não são necessários de início. Entretanto, quando examinado em detalhes, um conceito usualmente contém um grupo coerente de atributos.

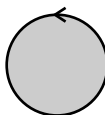
6.3.1 Atributos únicos

Assume-se que diferentes instâncias de um conceito seja distinguíveis uma da outra apenas pelo fato de que elas foram criadas independentemente. Este não é o caso com tipos primitivos. Dois valores como 05/03/2013 e 05/03/2013 são a mesma data. Mas dois compradores chamados Fulano de Tal não são necessariamente a mesma pessoa. Algumas linguagens de programação, inclusive, usam dois predicados de comparação, um para verificar se dois objetos são *iguais* (têm os mesmos valores para seus atributos) e outro para verificar se eles são *idênticos* (eles ocupam o mesmo espaço em memória).

Diferentes instâncias são, portanto, sempre distinguíveis uma da outra, mas adicionalmente elas podem ter atributos que são únicos para cada uma. Quando um atributo for estereotipado como <<*unique*>> duas instâncias do conceito não podem ter o mesmo valor para aquele atributo. Um exemplo de atributo único é o ISBN para os livros, o qual é único para cada livro (Figura 6.8), enquanto os outros atributos podem até ter valores idênticos para livros diferentes.


Figura 6.8

Classe mostrando um atributo estereotipado como <<unique>>.


Livir
Figura 6.9

Uma classe controladora de sistema.

Atributos únicos não devem ser confundidos com *chaves primárias*, <<pk>>. Chaves primárias são usadas especialmente no design de bancos de dados para fornecer uma identificação única para uma instância. Mas <<unique>> e <<pk>> não são a mesma coisa, porque um conceito só pode ter uma única chave primária, mas pode ter vários atributos únicos. Atributos únicos estão mais próximos da ideia de *chaves candidatas* em bancos de dados relacionais (Date, 1982).

6.3.2 Classe controladora de sistema

Usualmente, espera-se que o modelo conceitual seja um grafo conexo, ou seja, todo conceito tem um caminho (uma sequência de associações) que o conecta a qualquer outro conceito. Quando este não é o caso, supõe-se que algo esteja faltando, ou seja, associações ou conceitos importantes podem ainda não ter sido descobertos.

Embora alguns autores se oponham a isso, desde que alguns cuidados sejam tomados, pode ser útil ter no modelo conceitual um conceito que represente o sistema como um todo (em nosso exemplo, Livir). Primeiramente, este conceito deve ser declarado como um *Singleton*⁶; em segundo lugar, ele não deve ter atributos, sendo definido apenas como uma classe de controle. Essa classe corresponde ao *controlador de sistema* ou *controlador-fachada* já mencionado no Capítulo 5. Ele não representa quaisquer dados; serve apenas como ponto de referência para adicionar associações para outras classes, como mostrado nas próximas seções.

⁶ Um padrão de design que estabelece que uma classe com uma única instância pode ser acessível globalmente.

Todos os conceitos devem ser associados direta ou indiretamente ao controlador para serem acessíveis pela aplicação.

O controlador de sistema pode ser estereotipado com <<control>> ou desenhado segundo a notação icônica de Jacobson (1994), conforme mostra a Figura 6.9.

No Capítulo 9, as vantagens de usar uma classe controladora de sistema são explicadas em detalhe. Existem algumas situações que são modeladas de forma mais elegante com o uso da classe controladora⁷.

6.4 Associações

Quando conceitos complexos são relacionados, diz-se que existe uma *associação* entre eles. Associações usualmente ocorrem entre dois conceitos, mas elas também podem ocorrer entre três ou mais conceitos. Há também associações reflexivas que definem ligações entre instâncias de uma mesma classe. Por exemplo, a associação que liga os pais aos filhos é definida como associação reflexiva de *Pessoa* para *Pessoa*.

Quando classes são adicionadas, suas instâncias podem ser *ligadas*⁸. Por exemplo, um pedido está ligado aos livros que foram solicitados e também ao comprador que criou o pedido. Um pagamento, se existir, está ligado a um pedido no exemplo da livraria.

Associações podem ser difíceis de identificar nos textos dos casos de uso, especialmente porque algumas vezes decidir se algo é um conceito ou associação pode ser difuso. Por exemplo, uma *reserva de livro* é uma associação entre livro e comprador? Ou é um conceito? Se ela for considerada um conceito, quais são suas associações?

Além disso, os textos dos casos de uso frequentemente mencionam *operações* que não são exatamente associações estáticas. Operações tais como comprar livros ou fazer uma reserva são transformações dinâmicas sobre os dados. Elas poderiam produzir conceitos para representar sua existência, mas usualmente representá-las como associações não seria adequado. É necessário ter em consideração as diferenças entre as associações (estáticas) e as operações (dinâmicas):

- Uma *associação* é uma relação estática que pode existir entre conceitos complexos, complementando a informação sobre eles em um dado momento de tempo (seria uma fotografia de seu estado), ou referindo-se a informação associativa completamente nova (por exemplo, vizinhança ou relação pai/filho).
- Uma *operação* é o ato de acessar e transformar a informação existente.

Por ser uma descrição de um processo dinâmico, o texto do caso de uso usualmente está cheio de referências para operações, mas associações podem ser difíceis de inferir a partir do texto.

Em uma loja de carros usados, existem pessoas que comprem carros de outras pessoas. Quando alguém compra um carro, esta pessoa está realizando uma operação. Essa operação

⁷ Veja também um exemplo na Figura 6.18.

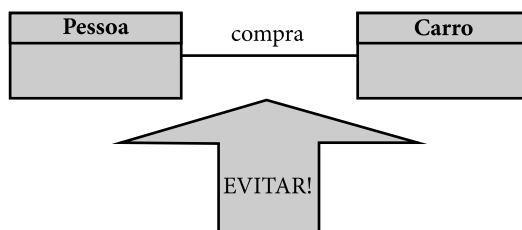
⁸ *Linked*.

muda as ligações entre os objetos: uma ligação de dono deixa de existir enquanto outra é criada em seu lugar. Não seria apropriado representar uma operação como associação, como mostrado na Figura 6.10.

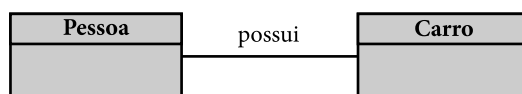
Por que esta interpretação é inadequada? Porque *comprar* não é um estado da informação, é uma transição entre estados.

O estado da informação sobre carros e pessoas, neste caso, é a posse: uma dada pessoa pode possuir um carro ou não. Isso pode ser representado como uma associação entre *Pessoa* e *Carro* (Figura 6.11).

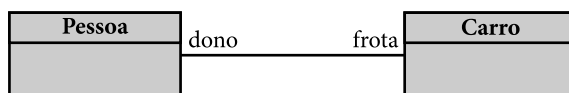
Entretanto, existem diferentes formas pelas quais pessoas e carros podem estar associados, além da relação de posse. Uma pessoa pode ser *passageira* de um carro, ou seu *motorista*, ou ainda a associação pode representar simplesmente que alguém *gosta* de um determinado carro. Para eliminar essas ambiguidades, é conveniente, em muitos casos, usar um *nome de papel* em um ou ambos os lados de uma associação para indicar qual o papel que cada classe executa em relação à outra. Assim, uma associação binária tem dois nomes de papel: um para cada classe participante. Na Figura 6.12, por exemplo, uma pessoa no papel de *dono* se relaciona com um carro e um carro pertence ao papel *frota* de uma pessoa.


Figura 6.10

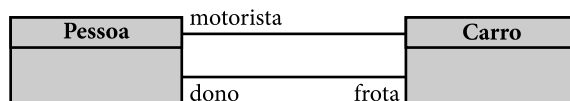
Uma operação inadequadamente rotulada como associação.


Figura 6.11

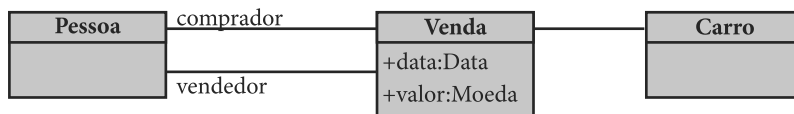
Representação de uma associação estática entre dois conceitos.


Figura 6.12

Uma associação com nomes de papéis.

**Figura 6.13**

Múltiplas associações entre os mesmos conceitos.

**Figura 6.14**

Uma transação representada como um conceito.

Diferentes papéis podem ser representados por diferentes associações, como mostra a Figura 6.13.

Quando um nome de papel está ausente do diagrama, assume-se que o nome da classe (iniciando em minúscula) seja o nome *default* do papel. No caso da Figura 6.13, um carro tem dois papéis em relação a pessoas: *dono* e *motorista*. Do ponto de vista de uma pessoa, há dois papéis para carros: a *frota*, que corresponde aos carros que uma dada pessoa possui, e um papel *default* chamado *carro*, que corresponde aos carros que a pessoa dirige.

De acordo com a especificação da UML, associações também podem ter nomes (a Figura 6.11 apresenta um exemplo), o qual é colocado no meio da linha que liga as classes. Entretanto, tais nomes, além de ser difíceis de definir (analistas novatos costumam encher seus diagramas com várias associações rotuladas com “possui” ou seus sinônimos), não têm muita utilidade prática.

Pode ser mais fácil e mais produtivo trabalhar com nomes de papéis em vez de nomes de associação. Na prática, pode-se até ignorar o fato de que associações podem ter um nome. Modelos conceituais podem ser perfeitamente claros com nomes de papel substituindo os nomes de associação. Nomes de papel são mais fáceis de dar, porque eles definem possibilidades de navegação entre conceitos e são úteis também para nomear elementos de código de programação deles derivados, como nomes de variáveis que representariam a associação quando o código fosse gerado. Se, por algum motivo, os nomes de associação ainda fossem necessários, sempre é possível criar um nome de associação pela composição dos nomes dos dois papéis; o inverso não é necessariamente verdade. Por exemplo, uma associação entre um livro e um comprador poderia ser chamada de *livro_comprador* se for necessário que ela tenha um nome (por exemplo, para ser referenciada como uma entidade de primeira classe).

Algumas vezes, pode ser interessante manter informação sobre operações ou transações que foram realizadas. Por exemplo, poderia ser útil para o sistema de vendas de

carros armazenar informações sobre transações de venda, como, por exemplo, data, valor etc. Nesse caso, a informação sobre a transação é incluída no diagrama como um conceito complexo (Figura 6.14), e ela representa estaticamente a operação que foi realizada.

Nesse caso, a transação é representada como um conceito, enquanto suas associações representam relações estáticas entre conceitos.

6.4.1 Multiplicidade de papel

Em um modelo conceitual, é fundamental saber quantos elementos um papel aceita. Por exemplo, na associação entre *Pessoa* e *Carro* com o nome de papel *motorista* na Figura 6.13, quantos carros alguém pode dirigir? Quantos motoristas um carro pode ter?

A resposta depende de um estudo sobre a natureza do problema e do real significado da associação, especialmente sobre se ela representa o presente ou o passado. Por exemplo, se a associação da Figura 6.13 representa o presente, então um carro só pode ter no máximo um motorista (um de cada vez). Mas se a associação representa o passado, então é possível que um carro tenha vários motoristas que o dirigiram em diferentes períodos de tempo.

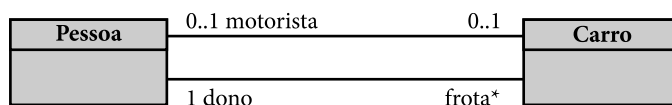
Assim, é fundamental que a equipe decida claramente o que a associação significa antes de decidir sobre sua multiplicidade de papel. Ela precisa lembrar que associações são restrições estáticas; elas representam as ligações que podem existir entre as instâncias de classes associadas. Portanto, dependendo do significado da associação, ela pode ter diferentes multiplicidades de papel.

Existem basicamente duas decisões a serem tomadas sobre a multiplicidade de um papel de associação:

1. O papel é obrigatório ou não? Por exemplo, uma pessoa deve ter pelo menos um carro? Um carro deve ter pelo menos um motorista ou proprietário?
2. O número de instâncias que podem ser ligadas por meio do papel tem um limite conceitual definido? Por exemplo, existe um número máximo ou mínimo de carros que uma pessoa pode dirigir ou possuir?

As respostas a essas questões podem ser complicadas. Em relação à primeira decisão, por exemplo, retornando ao projeto *Livir*, espera-se que todo pedido tenha um pagamento. Mas isso não qualifica o papel do pagamento como obrigatório, porque um pedido pode existir sem um pagamento por certo período de tempo. Eventualmente, qualquer pedido será – assim se espera – pago, mas não necessariamente logo. Assim, este papel *não* é obrigatório para o pedido.

Outra dificuldade refere-se ao número máximo de elementos que um papel permite. Fisicamente falando, o número máximo de carros que uma pessoa pode ter é o número de carros que existe no planeta Terra (número que está aumentando, mas poderia ser definido em um dado momento de tempo, mesmo se fosse muito difícil de contar). Mas à medida que novos carros possam ser construídos, eles também poderiam ser comprados. Assim, embora haja uma limitação física para o papel de dono, não há uma limitação lógica para ele. Assim, o papel deveria ser considerado virtualmente sem limite superior.

**Figura 6.15**

Exemplo de associações com multiplicidade de papel.

A multiplicidade de papel em UML é representada por uma expressão numérica onde:

- O asterisco (*) significa “qualquer quantidade” e indica que não há limite superior.
- A vírgula (,) significa “ou”.
- Dois pontos (..) significa “até”.

Seguem exemplos de limites de multiplicidade para modelos conceituais:

1 – exatamente um

0..1 – zero ou um

* – zero ou mais

1..* – um ou mais

2..5 – de dois até cinco

2,5 – dois ou cinco

2,5..8 – dois ou de cinco até oito

Os limites de multiplicidade que incluem o zero representam papéis que são opcionais para um conceito. Todos os outros limites são obrigatórios.

A Figura 6.15 mostra um exemplo no qual uma pessoa pode ter uma frota com um número não determinado de carros (opcional), e uma pessoa pode dirigir um ou mais carros (também opcional). No entanto, ele mostra que um carro deve ter um único dono (obrigatório) e que ele pode ter um motorista ou não (opcional).

De acordo com as leis de muitos países, um carro pode ter mais do que um dono ou pode ser possuído por uma corporação; em algumas situações, um carro até poderia não ter dono algum. O modelo conceitual (tal como mostrado na Figura 6.15) não representa necessariamente uma verdade geral, mas as necessidades de informação para uma dada aplicação. Para alguns sistemas de informação, poderia ser obrigatório que um carro, para ser registrado, tenha um único dono e que este dono deve ser uma pessoa. Nesse caso, o modelo conceitual deveria representar as necessidades de informação pragmáticas e específicas para este sistema.

6.4.2 Direção das associações

Uma associação em um modelo conceitual deveria ser *não direcional*, isto é, a navegabilidade da associação ainda não deveria ser determinada (Booch *et al.*, 2007). Isso é em razão do fato de que, para os propósitos da análise, é necessário apenas saber se dois conceitos estão associados ou não, o projeto dinâmico é que vai decidir sobre navegabilidade mais tarde.

Existe, usualmente, pouco benefício em uma definição prematura (antes do design dinâmico) da direção de navegabilidade. Associações *unidirecionais* em geral têm as seguintes vantagens sobre as bidirecionais:

- Não é necessário definir um nome de papel para ambos os lados (apenas um deles).
- Elas são mais fáceis de implementar.

Apenas a primeira vantagem tem algum significado para a modelagem conceitual, mas ela é muito fraca para justificar a tomada prematura de decisões.

6.4.3 Associação derivada

Tão úteis quanto os atributos derivados são as *associações derivadas*, isto é, associações que, em vez de serem representadas fisicamente, são calculadas a partir da informação disponível. Por exemplo, suponha que é frequentemente necessário conhecer quais livros um dado comprador efetivamente comprou no sistema Livir. Como os livros não estariam diretamente ligados ao comprador no modelo conceitual, referir-se a essa informação poderia ser um pouco mais complicado do que o esperado. A Figura 6.16 mostra um exemplo de como um modelo conceitual para Livir poderia relacionar livros e compradores indiretamente por meio dos pedidos e itens de compra; nesse caso, o conjunto de todos os livros comprados pelo comprador seria referenciado como *o conjunto de livros ligados aos itens ligados aos pedidos ligados ao comprador*.

Comparado ao modelo conceitual preliminar do Capítulo 3, a Figura 6.16 introduz o conceito de *Item* entre o *Livro* e o *Pedido*. Itens são necessários porque o conceito de *Livro* representa o título publicado, não a cópia física. Nesse caso, o que é solicitado é um *Item*, o qual representa uma ou mais cópias físicas de um dado *Livro*. O livro pode ter seu preço regularmente atualizado, mas a mudança no preço de um livro não deveria afetar o preço dos pedidos antigos. Assim, o preço de um item deve permanecer o mesmo, ainda que o preço do livro tenha mudado depois que o pedido foi feito.

Note que com este modelo não há nenhuma associação direta entre comprador e os livros que ele já comprou. Criar uma nova associação direta entre *Comprador* e *Livro* poderia não ser uma boa solução, porque isso poderia ser redundante no modelo, uma vez que o conjunto de livros comprados já pode ser obtido indiretamente. Além disso, uma nova associação entre *Comprador* e *Livro* poderia ligar *qualquer* comprador com *qualquer* livro, não apenas aqueles que já foram comprados por meio dos pedidos. Este modelo permitiria a representação de informação inconsistente, a qual poderia requerer algum mecanismo de verificação (por exemplo, permitindo que apenas livros que já foram comprados por ele sejam associados diretamente ao comprador).

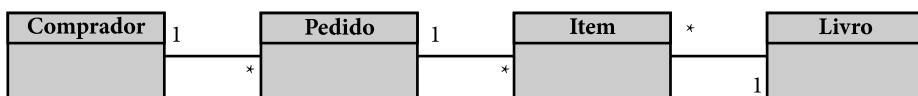
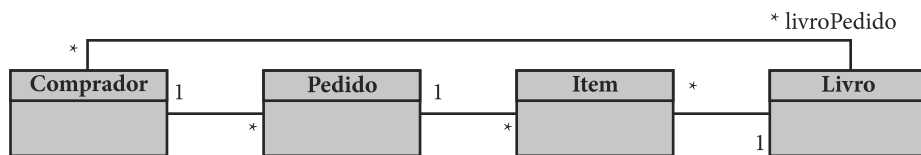


Figura 6.16

Um comprador indiretamente associado com livros.

**Figura 6.17**

Uma associação derivada.

Uma solução de modelagem para este caso, quando for relevante ter acesso direto para uma coleção de objetos que já pode ser derivada da informação existente, é usar uma *associação derivada*, como mostrado na Figura 6.17.

UML permite que associações derivadas sejam navegadas nos dois sentidos. Assume-se, portanto, que ambos os papéis são derivados. No caso da Figura 6.17, o papel oposto *comprador* deveria também ser considerado derivado, mesmo que seu nome de papel não fosse indicado no diagrama.

A multiplicidade de ambos os papéis da associação derivada da Figura 6.17 é para muitos porque, navegando pelas associações normais, pode-se ver que um comprador pode ter pedido zero ou mais livros, e que um livro pode ter sido pedido por zero ou mais compradores.

A multiplicidade de uma associação derivada depende de ela aceitar repetição de elementos ou não (ver Seção 6.5). Em ambos os casos, o *limite superior* da multiplicidade derivada será o *produto* dos limites superiores dos papéis no caminho de navegação. Já no caso do *limite inferior*, deve-se observar o tipo de associação:

- No caso de associações que *não admitem a repetição* de elementos dentro do papel (conjuntos e conjuntos ordenados), o limite inferior da multiplicidade derivada é o *maior* dentre os limites inferiores dos papéis no caminho de navegação.
- No caso de associações que *admitem a repetição* de elementos dentro do papel (*bags* e listas), o limite inferior da multiplicidade derivada é o *produto* dentre os limites inferiores dos papéis no caminho de navegação.

Por exemplo, se os papéis não admitem repetição de elementos e as multiplicidades encontradas no caminho são 2..5, 3..* e 1 então a multiplicidade derivada é 3..*, porque os limites inferiores são 2, 3 e 1 (o maior dentre eles é 3), e os limites superiores são 5, * e 1, e o produto de qualquer número por infinito é infinito, portanto o limite superior é *.

No entanto, se os papéis admittissem a repetição de elementos, então o limite inferior seria $2 * 3 * 1 = 6$, e a multiplicidade derivada seria 6..*.

Entretanto, se filtros fossem aplicados à definição da associação derivada, a multiplicidade resultante poderia ainda ser diferente. Por exemplo, se a associação derivada fosse definida de forma que ela liga um comprador apenas ao livro mais caro que ele já comprou (considerando que ele já tenha comprado pelo menos um livro), então a multiplicidade derivada seria 1 e não *.

Ao contrário das associações normais que permitem que ligações entre objetos individuais sejam adicionadas e removidas, uma associação derivada pode apenas ser consultada (similarmente ao que ocorre com atributos derivados).

Uma associação derivada também pode ser definida por uma expressão OCL. O exemplo da Figura 6.17 pode ser definido como:

```
Context Comprador::livroPedido
  derive:
    self.pedido.item.livro
```

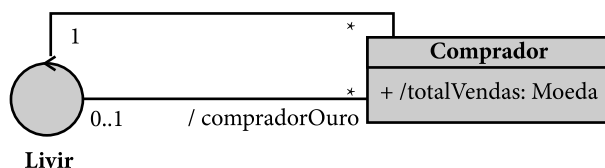
As seguintes observações podem ser feitas sobre essa expressão OCL:

- O contexto é a classe na origem da associação derivada (*Comprador*). O subcontexto é o nome de papel da associação derivada (*livroPedido*).
- A cláusula *derive*: é usada de forma similar ao caso dos atributos derivados. O que determina se a expressão está definindo um atributo ou associação derivada é o contexto e o tipo de informação que é retornado pela expressão. Atributos derivados são definidos por expressões que retornam um tipo alfanumérico, enumeração ou primitivo, enquanto associações derivadas são definidas por expressões que retornam um objeto ou coleção de objetos (estrutura de dados).

Em linguagens orientadas a objetos, usualmente a notação “.” pode ser usada apenas sobre um objeto único. OCL, entretanto, permite que ela seja usada sobre coleções de objetos⁹. Se x é uma coleção, então o significado de $x.y$ é a coleção dos y que podem ser obtidos como propriedades de x . Por exemplo, *self.pedido* é o conjunto de pedidos de um comprador no exemplo corrente; *self.pedido.item* é o conjunto dos itens ligados aos pedidos de um dado comprador; e *self.pedido.item.livro* é o conjunto de todos os livros ligados aos itens, que estão ligados aos pedidos e que estão ligados a um dado comprador.

Uma associação derivada pode também ser usada para definir um subconjunto de objetos a partir de um conjunto de objetos ligados por meio da aplicação de um filtro. Por exemplo, imagine que a livraria tenha um registro de compradores especiais que já compraram mais de R\$ 1.000,00 em livros. Vamos chamá-los de *compradores ouro*. Vamos também assumir que a classe *Comprador* tem um atributo derivado *totalVendas* com a soma de todos os pedidos do comprador. A definição desse atributo derivado não é importante para o exemplo e será ignorada por ora. Uma associação derivada entre o controlador e a classe *Comprador* pode ser usada para modelar os compradores ouro como um subconjunto dos compradores de Livir. A Figura 6.18 mostra o modelo conceitual parcial para este exemplo.

⁹ De fato, em OCL, mesmo um objeto único é considerado uma coleção com um único elemento. Isso é equivalente à noção natural de um conjunto, mas não equivalente à noção matemática de conjunto, porque, na teoria dos conjuntos, um conjunto com um elemento não é o elemento. Em OCL, um conjunto com um elemento e o elemento são a mesma coisa.

**Figura 6.18**

Associação derivada definindo um subconjunto de uma associação normal.

A expressão OCL para esta associação derivada tem como contexto a classe *Livir* porque ela se origina nesse controlador. A expressão que define a associação derivada tem de indicar o conjunto de compradores que compraram mais de R\$ 1.000,00, ou seja, as instâncias de *Customer* para as quais o *totalVendas* é maior do que 1000.

A expressão OCL que obtém um subconjunto de elementos que satisfaz uma dada expressão Booleana como esta é *select*. A expressão Booleana é colocada entre parênteses após o comando *select*. A expressão Booleana usualmente inicia com uma variável de iteração, que no caso da expressão seguinte foi nomeada como *umComprador*:

```
Context Livir::compradorOuro
  derive:
    self.comprador->select (umComprador |
      umComprador.totalVendas>1000
    )
```

No caso dessa expressão, a variável de iteração é nomeada como “*umComprador*” mas ela poderia ter qualquer outro nome. Ela denota cada um dos compradores para a avaliação da expressão *umComprador.totalVendas>1000*.

A notação de seta (->) é usada antes do *select*, em vez da notação ponto (.), porque a função *select* é aplicada sobre a estrutura do conjunto e não sobre seus elementos¹⁰.

OCL também permite que a variável de iteração seja omitida se o contexto da expressão é não ambíguo. Assim, a expressão anterior poderia ser simplificada para:

```
Context Livir::compradorOuro
  derive:
    comprador->select (totalVendas>1000)
```

Observe que dentro do parênteses, após o *select*, o contexto é potencialmente ambíguo. Ali a propriedade *totalVendas* poderia se referir tanto a um comprador quanto a instância de *Livir* (*self*). O que nos permite eliminar essa ambiguidade é o fato de que a propriedade *totalVendas* só existe na classe *Comprador*; ela não existe na classe *Livir*. Se ela existisse em ambas as classes, então a notação não abreviada teria que ser usada para eliminar a ambiguidade.

¹⁰ No caso de um conjunto de pessoas *p*, se alguém quiser obter o conjunto de todas as idades, ele poderia escrever *p.idade*. Entretanto, o tamanho do conjunto não é uma propriedade das pessoas; ele é uma propriedade da *estrutura do conjunto*. Assim, para obter o tamanho do conjunto, a expressão *p->size()* deveria ser usada. Se fosse escrita como *p.size* ela estaria representando o conjunto dos “tamanhos” de cada pessoa, se tal atributo existisse, e não o tamanho do conjunto.

6.4.4 Agregação e composição

Algumas associações podem ser consideradas *mais fortes* do que outras no sentido de que elas definem que um tipo de objeto é composto de outros. Uma casa, por exemplo, é composta de seus cômodos; um livro, de seus capítulos; e um curso de graduação, de suas disciplinas.

Se a associação é exclusiva, no sentido de que um objeto não pode ser parte de outro todo ao mesmo tempo, então ela é considerada uma associação parte-todo forte, a qual é chamada de *composição*¹¹, e é desenhada como um diamante negro no papel que representa o todo (Figura 6.19).

A multiplicidade do papel que representa o todo (a parte onde fica o diamante), no caso da composição deve ser 1 ou 0..1, e nenhuma outra, porque composições não podem compartilhar partes, mesmo com objetos da mesma classe.

Composição indica exclusividade na associação parte-todo. Quando esta exclusividade não é obrigatória (a parte pode ser parte de outras agregações), então se usa um diamante branco para indicar que a associação é uma *agregação* (Figura 6.20).

O diamante branco indica uma agregação compartilhada onde a parte pode pertencer a diferentes todos ao mesmo tempo. Na Figura 6.20, um dado *Item* pode ser parte de um *Pedido*, mas também pode ser parte de uma *Venda*.

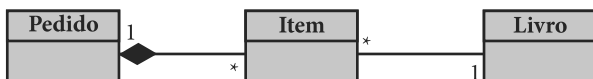


Figura 6.19

Um exemplo de composição.

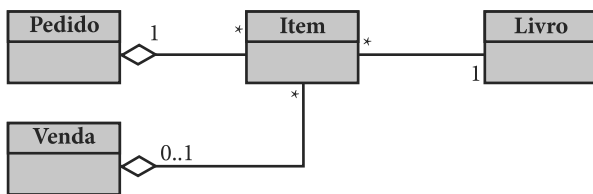


Figura 6.20

Um exemplo de agregação.

¹¹ “Uma associação pode representar uma composição (isto é, um relacionamento parte/todo). Apenas associações binárias podem ser agregações. Composições são uma forma forte de agregação que requer que parte da instância seja incluída em, no máximo, uma composição de cada vez. Se a composição for deletada, todas as suas partes são normalmente deletadas com ela. Note que uma parte pode (quando permitido) ser removida da composição antes que ela seja deletada e, assim, não ser deletada como parte da composição. Composições podem ser ligadas em um grafo direcionado acíclico com características de deleção transitivas; isto é, deletar um elemento em uma parte do grafo vai resultar na deleção de todo o subgrafo abaixo daquele elemento.” (Object Management Group, 2011 – tradução do autor)

A composição e a agregação compartilhada são associações especiais e devem ser usadas com muita parcimônia, ou seja, elas devem ser usadas somente quando a equipe tiver certeza de que um objeto é realmente *parte* de outro, e não apenas uma associação normal. Mesmo a especificação da UML (Object Management Group, 2011) indica que a semântica precisa de uma agregação compartilhada que varie entre aplicações e entre modeladores, e isso contraindica seu uso indiscriminado.

É comum ver a agregação e a composição serem abusadas nos modelos, quando objetos que não são parte-todo são ligados por este tipo de associação. Por exemplo, um comprador não é parte de um pedido, se não por outra razão qualquer, mas pelo fato de que o comprador é uma pessoa e um pedido não é uma coisa física, mas uma transação. Composição e agregação compartilhadas devem unir elementos de mesma natureza: físico com físico e abstrato com abstrato. Um pedido pode ser associado a um comprador, mas ele não é *feito* de compradores.

Existem poucas vantagens reais em usar agregação na modelagem conceitual. Esta é outra razão para minimizar ou mesmo abolir seu uso nos modelos conceituais. Entre as vantagens está o fato de que as partes das agregações ou composições usualmente têm atributos que são combinados e derivados no todo. Por exemplo, o valor total de um pedido é a soma dos seus itens; o peso de um pacote é a soma do peso de cada livro; quando um carro é vendido, todas as suas partes são vendidas também, e assim por diante. Entretanto, essas questões normalmente aparecem apenas durante o design.

6.4.5 Associações n-árias

A maioria das associações é binária, isto é, a maioria das associações tem apenas dois papéis. Entretanto, existem situações nas quais associações com três ou mais papéis podem ser necessárias. Em UML essas associações podem ser representadas com o uso de um diamante que conecta todos os papéis da associação, como mostra a Figura 6.21, a seguir.

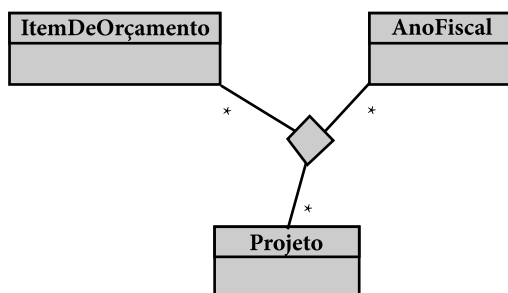


Figura 6.21

Um exemplo de associação ternária.

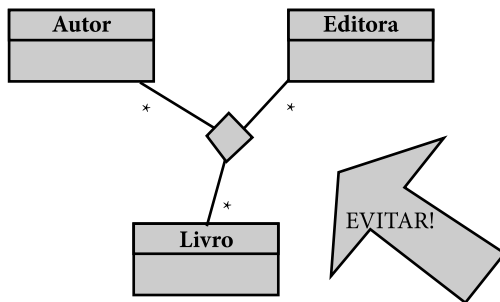


Figura 6.22

Uma associação ternária inadequada.

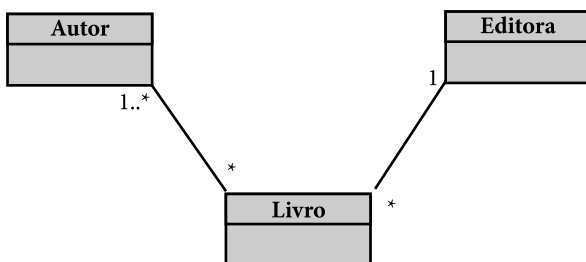


Figura 6.23

O modelo da Figura 6.22 corretamente representado com duas associações binárias.

O modelo representado na Figura 6.23 é mais preciso do que o da Figura 6.22 para o caso em discussão. Apenas em um cenário onde o mesmo autor pudesse publicar o mesmo livro usando diferentes editoras e uma editora pudesse publicar o mesmo livro com diferentes autores (!) seria o caso de uma associação ternária.

O exemplo da Figura 6.21 foi tomado de uma aplicação de controle de orçamentos. Cada projeto pode estar associado a múltiplos itens de orçamento e múltiplos anos fiscais. Cada ano fiscal pode ter vários projetos e vários itens. Uma associação ternária (isto é, uma associação com três papéis) representa esta situação, como mostrado na Figura 6.21.

Esses casos não são frequentes, mas eles podem aparecer em um projeto. Antes de decidir usar uma associação ternária ou n -ária (o que pode se tornar uma dor de cabeça durante o design e implementação), a equipe deveria verificar se ela não está confundindo a situação com um caso onde uma combinação de associações binárias deveria ser usada. Por exemplo, o caso que a Figura 6.22 mostra não está corretamente modelado como uma associação ternária, porque a relação entre o autor e a editora acontece apenas quando o autor publica um livro com a editora. Então, a associação entre o autor e a editora deveria ser derivada.

Associações ternárias reais, entretanto, ainda podem ser substituídas por uma combinação de associações binárias. Talvez a abordagem mais simples para realizar isso seja trocar a relação ternária (ou n -ária) por um novo conceito. Algumas vezes, dar nome a esse novo conceito pode ser complicado, mas pode ser feito. Por exemplo, o modelo da Figura 6.21 poderia ser substituído pelo da Figura 6.24.

Entretanto, existe uma diferença semântica sutil entre os modelos das Figuras 6.21 e 6.24. No caso da Figura 6.24, pode haver duas ou mais instâncias de *ItemDeDespesa* com exatamente os mesmos item de orçamento, ano fiscal e projeto. Isso não pode acontecer no caso da Figura 6.21. Assim, para que esta simplificação tenha a mesma semântica da associação ternária, é necessário complementá-la com uma invariante que impeça essa duplicidade (veja Seção 6.7).

6.5 Coleções

Simple coleções de objetos que não tenham nenhuma estrutura ou significado particular não devem ser, em princípio, representadas como conceitos, mas como associações. Qualquer papel de associação *já representa* uma coleção de objetos da classe associada. Assim, considerando novamente o exemplo da Figura 6.15, *frota* é um papel que associa um conjunto de carros a uma pessoa. O modelo apresentado na Figura 6.25 é, portanto, desnecessário e redundante.

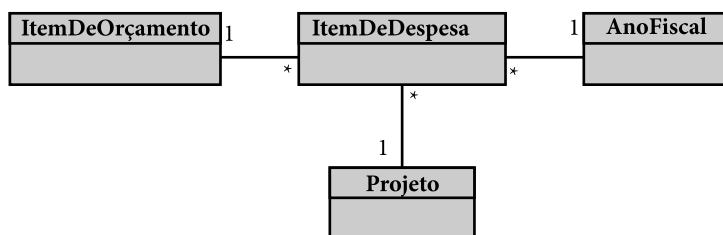


Figura 6.24

Modelo no qual uma associação ternária é trocada por um novo conceito.

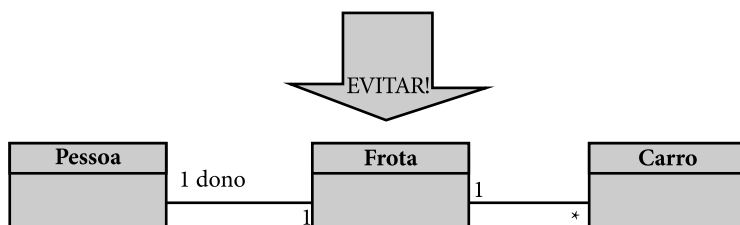
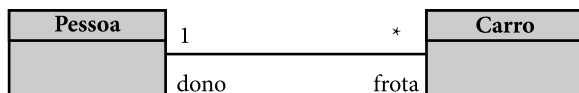


Figura 6.25

Uma simples coleção sendo inadequadamente representada como uma classe.


Figura 6.26

Um modelo mais simples para a situação apresentada na Figura 6.25.

Assim, a não ser que a frota tivesse seus próprios atributos ou associações (por exemplo, uma pessoa poderia ter várias frotas localizadas em diferentes lugares), a forma correta de representar uma simples coleção de objetos é por meio de um papel de associação (Figura 6.26) e não por meio de uma classe.

Porém, a equipe deve estar atenta ao fato de que se a coleção tiver atributos ou associações particulares, então ela deve ser representada como um conceito. Por exemplo, um *Pedido* pode ser inicialmente entendido como um simples conjunto de itens, mas um *Pedido* tem seus próprios atributos como data de emissão, valor total, desconto etc., e suas próprias associações como, por exemplo, com o comprador, o qual pode ter múltiplos pedidos. Se este for o caso, a coleção será um conceito. Entretanto, uma coleção sem nenhum atributo ou associação próprios é apenas uma coleção e, portanto, modelada como um papel de associação.

Papéis de associação podem ser pensados como representando *tipos abstratos de dados* (Liskov, 1974) em modelo conceituais. Em modelos de design, eles também poderiam representar tipos *concretos* de dados. Tipos abstratos de dados são definidos apenas pelo seu comportamento, não pela sua estrutura. Seguem alguns exemplos:

- A estrutura de *conjunto*, na qual elementos não se repetem e não têm posição definida.
- A estrutura de *bag*¹², na qual elementos podem se repetir, mas não têm posição definida.
- A estrutura de *conjunto ordenado*, na qual elementos não podem se repetir, mas têm posição definida.
- A estrutura de *lista* ou *sequência*, na qual elementos podem se repetir e têm posição definida.

Os tipos de dados *concretos*, no entanto, são implementações físicas das versões abstratas. Por exemplo, uma lista pode ser implementada de várias formas, tal como *array*, árvore binária, lista encadeada etc. No caso dos tipos concretos de dados, além do comportamento, há também uma estrutura física que os define.

6.5.1 Conjunto

Um papel de associação, por *default*, é um *conjunto*, uma estrutura na qual elementos não se repetem e não têm posição definida. Na Figura 6.26, a *frota* é um exemplo de conjunto.

Se alguém tentar ligar o mesmo carro com a mesma pessoa uma segunda vez, nada deve acontecer, porque o carro já pertence ao conjunto.

¹² Multiconjunto.

6.5.2 Conjunto ordenado

Suponha que um livro não esteja ainda em estoque, mas que exista um grupo de pessoas interessadas em comprá-lo assim que estiver disponível. Se não for possível garantir que será recebida uma quantidade de livros que permita atender a todos os pedidos, o mais justo seria atender a estes compradores estabelecendo uma fila desde a reserva mais antiga até a mais nova. Isso requer que os compradores sejam colocados em posições específicas na fila.

Se o mesmo comprador não puder reservar o mesmo livro mais de uma vez, então nessa estrutura ainda não existe repetição de elementos, mas eles têm posição definida. Isso leva à estrutura de *conjunto ordenado*. No diagrama, o papel deve ser marcado com a restrição *{ordered}* para indicar que é uma coleção na qual elementos não se repetem, mas têm posições definidas, como mostrado na Figura 6.27.

A restrição *{ordered}* sobre um papel de associação significa que os elementos naquele papel têm posição (primeiro, segundo, terceiro, ... último), mas não podem se repetir.

6.5.3 Bag

Há situações nas quais a posição dos elementos não importa, mas cada elemento pode aparecer mais do que uma vez na coleção. Essa estrutura é chamada de *bag* ou *multiconjunto*.

Por exemplo, alguém pode querer saber quantas pessoas visualizaram os detalhes de um livro, e pode ser importante também saber quantas vezes cada pessoa visualizou cada livro. Não é necessário saber quem visualizou primeiro, mas a informação do número de vezes é necessária. Esse é um caso típico para usar um *bag*. Cada vez que uma pessoa visualiza os dados de um livro, uma nova ligação é adicionada à associação definida como *bag*. O modelo é mostrado na Figura 6.28.

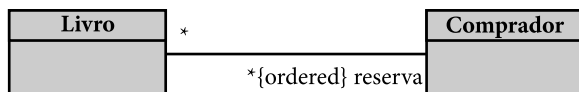


Figura 6.27

Um conjunto ordenado de reservas para um livro.

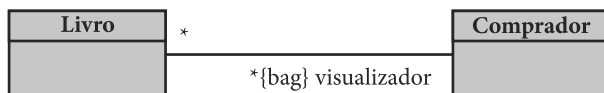


Figura 6.28

Um exemplo de *bag*.

6.5.4 Sequência

A *sequência* permite que elementos sejam repetidos e também atribui posição a cada um deles. Por exemplo, imagine que pessoas que compraram um determinado valor em livros se qualificam para receber um brinde, mas uma quantidade limitada de brindes está disponível a cada momento. Assim, a lista de pessoas que se qualificam para receber o brinde poderia ser construída e os brindes distribuídos aos primeiros membros da lista assim que os brindes estivessem disponíveis. Se alguém comprar novamente mais do que o limite estabelecido para receber o brinde, então essa pessoa vai entrar na lista de novo. Assim, nessa estrutura, a posição de um elemento é relevante e cada elemento pode aparecer mais do que uma vez: isso corresponde a uma estrutura de sequência, como mostrado na Figura 6.29.

A multiplicidade $*$ no lado *Livir* da associação significa que nem todos os compradores estão qualificados para receber um brinde e que eles podem se qualificar mais do que uma única vez.

A sequência tem dois casos especiais importantes. Quando o primeiro elemento a ser removido é sempre o mais antigo, como no caso da Figura 6.29, então a estrutura é também uma *fila*. UML não define uma restrição específica para filas, mas um estereótipo sobre a restrição da sequência poderia fazer este trabalho: `{sequence}<<queue>>`.

Quando o primeiro elemento a ser removido é o último a ser inserido na sequência, então ela se comporta como uma *pilha*, e pode ser estereotipada como tal: `{sequence}<<stack>>`. Filas e pilhas que não repetem elementos podem ser definidas respectivamente como `{ordered}<<queue>>` e `{ordered}<<stack>>`.

Uma lista genérica pode ter elementos removidos e inseridos em qualquer posição. Filas e pilhas têm elementos inseridos e removidos apenas em posições predeterminadas, como explicado anteriormente, embora, em alguns casos, todos os elementos possam ser visualizados.

Uma curiosidade a ser mencionada agora é que a maioria das coleções do mundo real são estritamente *conjuntos* e não *sequências*, mas, mesmo assim, a estrutura de programação mais usada é ainda a sequência e não o conjunto, o que indica a existência de um fosso entre a realidade e as estruturas de dados usadas em programação.

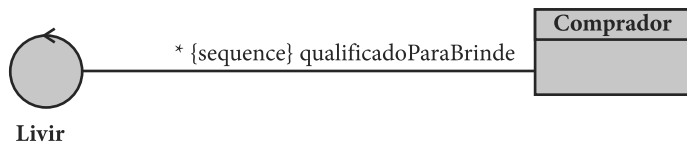
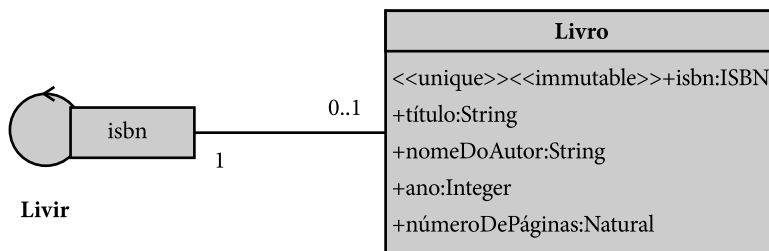


Figura 6.29

Um exemplo de sequência.

**Figura 6.30**

Um exemplo de associação qualificada representando um mapeamento.

6.5.5 Mapeamento

Quando um conceito tem um atributo único, um *mapeamento* pode ser criado a partir do valor daquele atributo para as instâncias do conceito. Por exemplo, como o ISBN é único para um livro, as associações de *Livro* poderiam ser qualificadas pelo valor do ISBN, significando que, em vez de um simples conjunto de livros, a classe de origem teria acesso a um mapeamento que permite identificar imediata e individualmente um livro qualquer desde que seu ISBN seja conhecido. A Figura 6.30 mostra um exemplo de associação qualificada.

O retângulo pequeno no lado esquerdo da associação na Figura 6.30 representa um *qualificador* para a classe no lado oposto da associação. Note que em vez do asterisco (*), o papel do lado direito está agora rotulado com “0..1”; a associação agora deveria ser lida como: “para cada possível ISBN não há mais do que *um* livro”. Alguns valores válidos de ISBN podem ainda não ter sido associados a qualquer livro, porque eles ainda não foram registrados no sistema ou porque, embora sintaticamente válido, o número de ISBN ainda não foi atribuído a nenhum livro; entretanto, se um ISBN for atribuído a um livro, nenhum outro livro poderá compartilhá-lo. Na prática, uma associação qualificada ainda é uma associação “para muitos”, mas os objetos pertencentes ao papel da associação podem ser identificados individualmente por um de seus atributos.

O papel do lado esquerdo da associação (o lado do qualificador) é marcado com “1”, significando que cada livro tem um único ISBN, nem menos nem mais. Isso tem de ser necessariamente verdadeiro, porque um ISBN é único e não opcional para a classe *Livro*, e, portanto, ele é obrigatório e não admite repetição.

O qualificador não é apenas uma forma de individualizar as instâncias ligadas ao papel da associação. Ele é também uma poderosa ferramenta de modelagem. Por exemplo, considere uma situação na qual uma pessoa possa ter vários telefones, mas apenas um de cada tipo. Como representar isso? Um modelo elegante para esta situação é mostrado na Figura 6.31.

No caso da Figura 6.31, uma pessoa pode ter até quatro telefones, mas apenas um de cada tipo. Se mais tipos forem adicionados à enumeração posteriormente, então mais telefones poderiam ser associados a cada pessoa. Se o papel do lado direito da associação fosse 1 em vez de 0..1, então uma pessoa deveria necessariamente ter quatro telefones, um de cada tipo. Este uso da associação qualificada com um papel obrigatório usualmente faz sentido apenas quando o tipo do qualificador é uma enumeração ou um conjunto finito.

No exemplo da Figura 6.30, o qualificador é um atributo da classe qualificada, e, nesse caso, ele pode ser chamado de *qualificador interno*. No entanto, na Figura 6.31, o qualificador não é um atributo da classe e assim pode ser chamado de *qualificador externo*.

6.5.6 Partição

Nas Figuras 6.30 e 6.31, a multiplicidade 1 ou 0..1 no lado direito da associação significa que a associação é um mapeamento, ou seja, para cada valor da chave existe no máximo uma instância da classe qualificada.

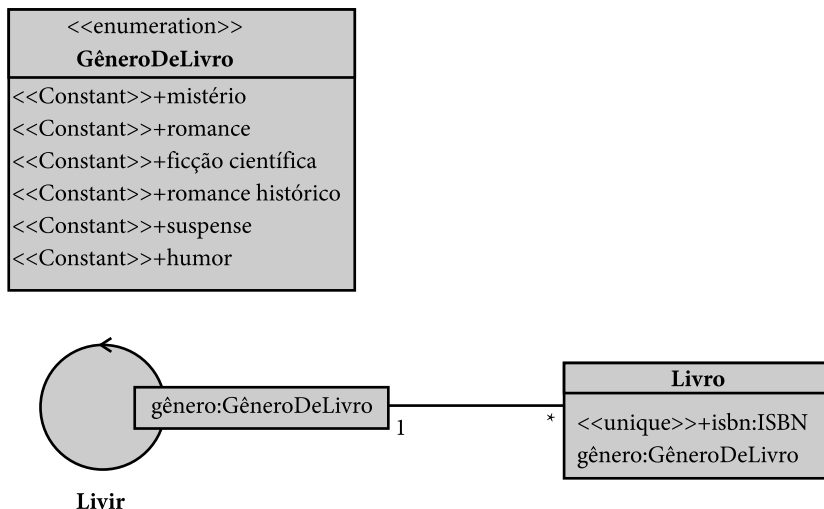
Mas se em vez de “1” ou “0..1” a multiplicidade fosse marcada com “*”? Nesse caso, a associação seria lida como segue: “para cada chave há um *conjunto* de instâncias da classe qualificada”.

Por exemplo, livros podem ser classificados pelo seu gênero. Como vários livros pertencem ao mesmo gênero, o atributo *gênero* não pode ser único. Mas é possível definir uma partição sobre conjunto de livros baseada no seu gênero, como mostrado na Figura 6.32.



Figura 6.31

Uso de um qualificador como ferramenta de modelagem.

**Figura 6.32**

Um exemplo de partição.

A Figura 6.32 estabelece que para cada valor particular de gênero há um conjunto de livros (com zero ou mais elementos). A multiplicidade de papel 1 no lado esquerdo indica que cada livro tem um único gênero.

6.5.7 Relação

Agora imagine que um livro possa ter mais de um gênero, como, por exemplo, o *Guia do Mochileiro das Galáxias* (Adams, 1979), o qual pode ser classificado tanto como ficção científica quanto como humor. Para representar a possibilidade de um livro ter mais do que um gênero no exemplo da Figura 6.32, teria sido suficiente mudar a multiplicidade do lado esquerdo da associação para * (se o gênero for opcional) ou 1..* (se for obrigatório cada livro ter pelo menos um gênero). Adicionalmente, o gênero teria de ser removido dos atributos de *Livro*, tornando-se um qualificador externo, porque o atributo *gênero* agora poderá aceitar múltiplos valores. A Figura 6.33 mostra a relação resultante.

A relação da Figura 6.33 estabelece que um livro pode ter um ou mais gêneros e um gênero tem um conjunto de livros associado a ele. Como um livro pode ter mais do que um gênero, então o qualificador deve ser necessariamente externo.

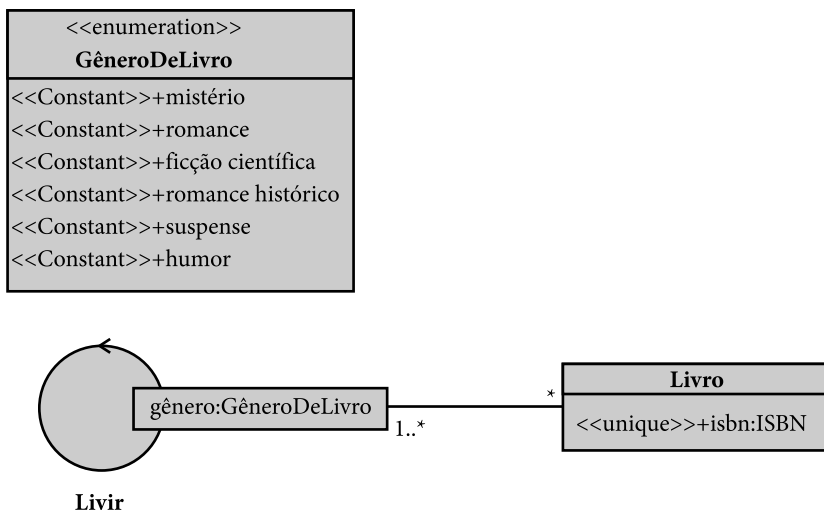


Figura 6.33

Um exemplo de relação.

6.6 Organização do modelo conceitual

A elaboração do modelo conceitual envolve mais do que simplesmente colocar conceitos, atributos e associações juntos. Para que o modelo conceitual seja uma representação confiável e organizada da informação do mundo real, é necessário usar algumas técnicas de modelagem.

As principais técnicas organizacionais para conceitos podem ser distinguidas em três grupos:

- *Estruturais*: representando generalizações entre conceitos, como, por exemplo, *ContaBancária* generalizando *ContaPoupança* e *ContaCorrente*.
- *Associativas*: representando papéis que alguns conceitos podem representar em relação a outros, como, por exemplo, *Estudante* e *Professor* sendo papéis que uma *Pessoa* pode representar em relação a uma escola.
- *Temporal*: representando relações entre diferentes estados de um conceito, como, por exemplo, um *Pedido* estando nos estados *PedidoPendente* e *PedidoPago*.

Analistas novatos e, algumas vezes, até os mais experientes tendem a pensar que a única forma de fatorar informação é pelo uso da herança ou generalização¹³. Entretanto, é necessário aplicar algum julgamento antes de decidir sobre usar essa ou aquela técnica. Herança só pode ser usada quando um conceito efetivamente tem dois ou mais subtipos. Embora UML permita que uma instância mude de classe, apenas umas poucas linguagens como Smalltalk (Goldberg; Robson, 1989) tentaram implementar esta característica, talvez

¹³ Tente, por exemplo, pesquisar no Google por imagens com as palavras “UML” e “herança”. Você verá vários exemplos de mau uso de herança, como, por exemplo, *Pessoa* sendo uma generalização de *Estudante* e *Professor*.

porque existam algumas dificuldades inerentes não resolvidas associadas com o fato de que uma instância possa mudar sua estrutura interna em tempo de execução. A maioria das linguagens orientadas a objetos ignora esta característica em razão de problemas que ela pode fazer surgir e também ao fato de que ela é usualmente desnecessária, porque muitos comportamentos que poderiam ser modelados com essas características também teriam representações alternativas mais simples. Assim, em termos práticos, usualmente é o caso de que uma instância não possa mudar sua classe: ela nasce como membro de uma classe e vai morrer como membro da mesma classe¹⁴. Como estudantes podem se tornar professores depois de formados, eles não se qualificam como subtipos de pessoas (a melhor solução para modelar essa situação é mostrada na Seção 6.6.2).

Outro contraexemplo é *Funcionário* e *Comprador* sendo modelados como subclasses de *Pessoa*. Isso é ruim, primeiramente, porque ninguém nasce comprador ou funcionário; uma pessoa *se torna* comprador ou funcionário quando se *associa* a uma empresa. Mais do que isso, a mesma pessoa pode ser comprador ou funcionário em *mais do que uma* empresa. Usar herança para este caso poderia criar problemas conceituais que podem produzir duplicidade nos registros e inconsistência nos dados.

As subseções seguintes apresentam detalhes sobre as três abordagens para organização de conceitos.

6.6.1 Generalização, especialização e herança

Por muitos anos, herança foi considerada a característica mais importante de linguagens orientadas a objetos. Linguagens como Smalltalk, por exemplo, foram construídas sobre uma única hierarquia de classes baseada em herança.

À medida que o tempo passou, essa ênfase perdeu força porque se percebeu que a herança não é sempre a melhor ideia para resolver problemas de modelagem. Hoje em dia, a herança é considerada apenas mais uma das técnicas que podem ajudar a fatorar a informação que de outra forma seria repetida nas classes.

Herança, em sistemas orientados a objetos, ocorre quando duas classes se relacionam por meio de uma associação especial chamada *generalização* (ou *especialização*, dependendo da direção em que se olha). Uma classe que generaliza outra classe é sua *superclasse* e a classe especializada é a *subclasse*.

Generalização deveria ser usada sempre que houver um conjunto de classes $X_1, \dots, X_n$ que tem diferenças específicas e similaridades em comum, de forma que as similaridades possam ser agrupadas em uma superclasse X (generalização de $X_1, \dots, X_n$) e as diferenças mantidas em $X_1, \dots, X_n$.

A generalização tem uma semântica muito diferente das outras associações do modelo conceitual. Ela existe apenas entre as classes, não entre as instâncias, como no caso das associações normais. Por exemplo, se uma classe *Comprador* está associada com a classe

¹⁴ Mais especificamente, um objeto também é instância de todas as suas superclasses (Seção 6.6.1). A ideia é que o conjunto de superclasses não possa mudar depois que a instância é criada.

Pedido por uma associação normal, então as instâncias de *Comprador* poderão estar ligadas a instâncias de *Pedido*: a associação normal é que permite que essas ligações existam. No entanto, se uma classe *ContaBancária* é uma generalização de *ContaCorrente*, então cada instância de *ContaCorrente* é *também* uma instância de *ContaBancária* (Figura 6.34). Não existem duas instâncias a serem ligadas: apenas uma única instância com as propriedades definidas em ambas as classes. No exemplo, toda instância de *ContaCorrente* têm três atributos: *número*, *limiteDoNegativo* e *taxaDoNegativo*. Entretanto, nem toda instância de *ContaBancária* é uma instância de *ContaCorrente*: a generalização é antissimétrica. Uma possível instância de *ContaBancária* teria apenas um atributo: *número*.

Assim, uma das razões para usar herança é quando existem algumas propriedades similares (atributos, associações ou métodos) em diferentes classes que podem ser fatorados em uma construção abstrata como uma superclasse.

Se a superclasse puder ter suas próprias instâncias, então ela é uma classe normal. Entretanto, muitas vezes, as superclasses são definidas de forma a não permitir que tenham instâncias diretas: apenas suas subclasses poderiam ter instâncias. Nesse caso, elas são definidas como superclasses *abstratas*. Superclasses abstratas são definidas em UML como uma classe cujo nome foi escrito em itálico ou, mais visivelmente, com a restrição {*abstract*} (Figura 6.35).

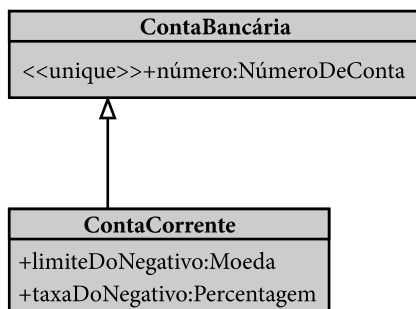


Figura 6.34

Um exemplo de generalização com herança.

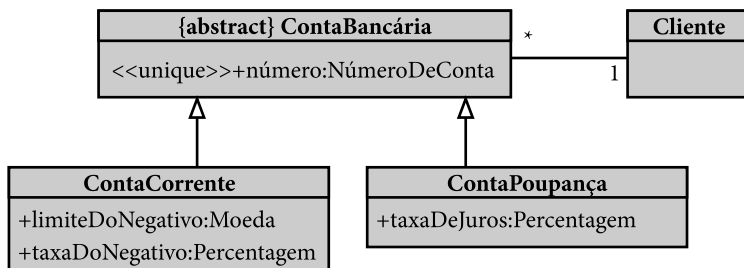


Figura 6.35

Duas classes sendo generalizadas por uma classe abstrata.

Na Figura 6.35, a classe *ContaBancária* é abstrata e, portanto, não pode ter instâncias diretamente; apenas suas subclasses podem ter instâncias. Instâncias de *ContaCorrente* têm como atributos: *número*, *limiteDoNegativo* e *taxaDoNegativo*; instâncias de *ContaPoupança* têm como atributos: *número* e *taxaDeJuros*. Ambas as subclasses também herdam a associação com a classe *Cliente* e todas as instâncias de ambas as classes devem, portanto, estar ligadas a uma instância de *Cliente*.

A generalização não é aconselhável quando a superclasse não tem propriedades, ou seja, não tem atributos, associações ou métodos¹⁵ próprios (Figura 6.36).

A generalização também não deve ser usada quando as subclasses não têm propriedades (atributos, associações ou métodos) que diferenciem uma da outra, como na Figura 6.37.

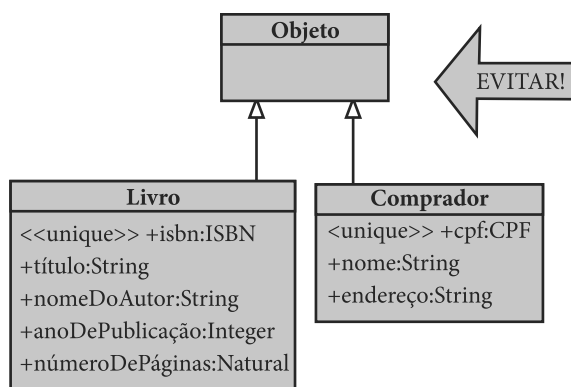


Figura 6.36

Situação na qual a generalização não é recomendada porque não há propriedades generalizadas.

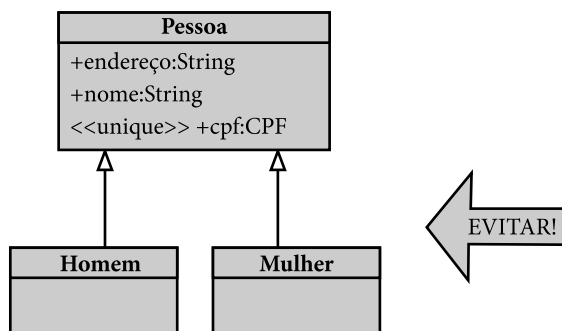


Figura 6.37

Situação na qual a especialização não é recomendada porque não há propriedades sendo especializadas nas subclasses.

¹⁵ Métodos somente são considerados quando classes de design forem modeladas. Lembre-se que as classes do modelo conceitual não têm métodos.

Nesses casos, a solução ideal é usar um atributo único, possivelmente tipado com uma enumeração para diferenciar grupos de instâncias de objetos que têm as mesmas propriedades (Figura 6.38).

Além dessas regras, antes de decidir sobre o uso da herança, é recomendável verificar se a generalização realmente representa uma classificação estrutural dos elementos, e não uma organização associativa ou temporal, como será mostrado nas próximas seções.

6.6.2 Classes de associação

Um tópico frequentemente mal entendido em modelagem conceitual relaciona-se com a definição e com a generalização entre classes que não são realmente subtipos estruturais, mas *papéis*. Por exemplo, uma livraria pode ter dois “tipos” de pessoas: compradores e trabalhadores. Após descobrir que ambos têm um nome, endereço, telefone etc., um analista poderia assumir que eles são dois tipos que poderiam ser generalizados como *Pessoa*. Mas essa solução aparentemente simples (Figura 6.39) gera um problema complicado, pois eles não são tipos diferentes de pessoa, mas diferentes papéis que pessoas podem desempenhar em relação a uma empresa.

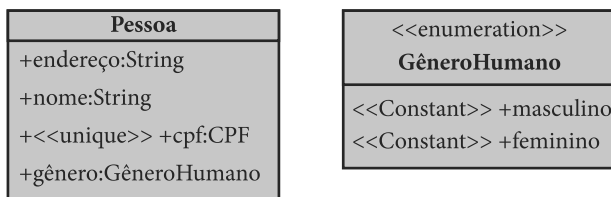


Figura 6.38

Um modelo mais adequado para a situação da Figura 6.37.

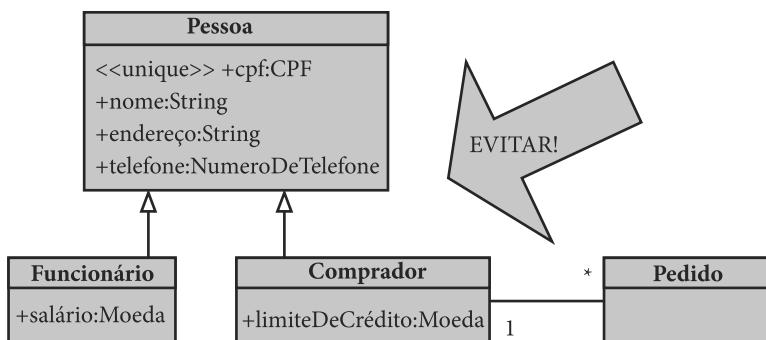


Figura 6.39

Forma inadequada de representar papéis como generalização.

Por que isso é um problema? Imagine que um trabalhador da livraria decide comprar livros no seu local de trabalho. Nesse caso, o trabalhador estará se comportando como um comprador. Uma solução inadequada, embora muito frequente nesse caso é criar um segundo registro para o funcionário como comprador, como se ele fosse uma pessoa diferente. Como consequência, a mesma pessoa terá dois registros no sistema: um como funcionário e outro como comprador. Isso produz redundância de dados e é uma fonte de inconsistência de dados, porque, por exemplo, se o funcionário registrou uma mudança de endereço, o comprador ainda vai ficar com o endereço antigo. Como eles são a mesma pessoa, essa informação está inconsistente.

Uma solução mais adequada seria considerar que existe uma *Pessoa* que pode se relacionar com uma *Empresa* de pelo menos duas formas: como comprador e como funcionário. As propriedades de um comprador (limite de crédito, por exemplo) e de um funcionário (salário, por exemplo) poderiam ser propriedades das associações, e não das pessoas. Para representar essas propriedades de associações, uma *classe de associação* poderia ser definida para cada uma, como mostrado na Figura 6.40.

Portanto, quando o mesmo objeto puder representar diferentes papéis em relação a outros objetos, esses papéis não devem ser representados como subclasses, mas como *classes de associação*.

Para decidir quais situações exigem herança e quais exigem classes de associação, pode-se verificar se os “subtipos” dependem da existência de uma terceira classe para fazer sentido. Se o “subtipo” depender de uma terceira classe, então a solução consiste em usar uma classe de associação. Por exemplo, ninguém pode ser simplesmente um estudante; se alguém é estudante, então ele deve estar associado a uma instituição de ensino ou pelo menos a um professor.

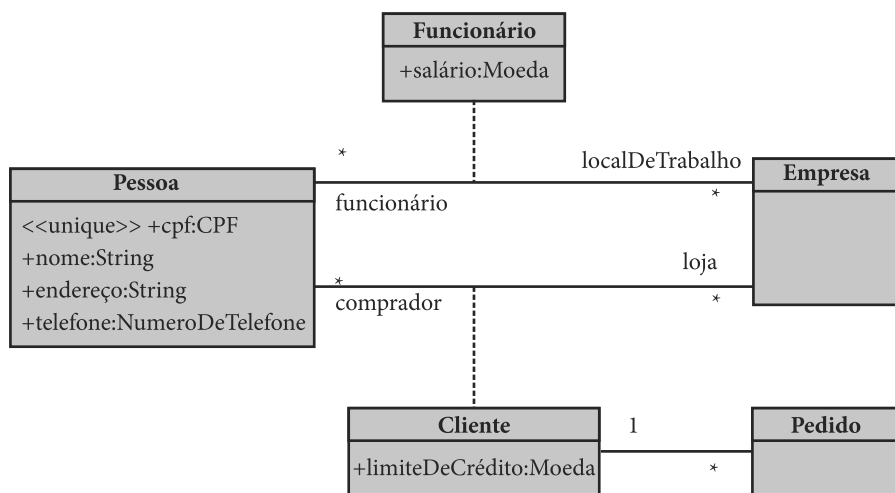
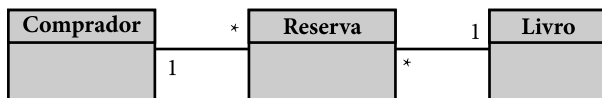
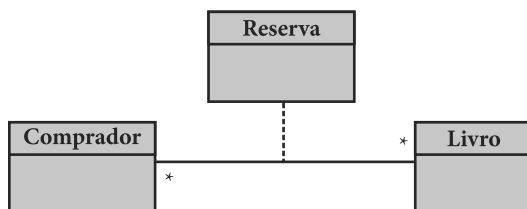


Figura 6.40

Forma mais adequada de representar papéis como classes de associação.


Figura 6.41

Uma reserva sendo modelada como um conceito intermediário.


Figura 6.42

Uma reserva sendo modelada como uma classe de associação.

Assim, seria inadequado criar classes que representam tipos de pessoas que, na realidade, não são subclasses, mas papéis. Funcionários, professores, estudantes, diretores, compradores etc. não podem ser subclasses de *Pessoa*. Conceitos como esses apenas fazem sentido se forem relacionados a outros conceitos tais como *Empresa*, *Escola*, *Departamento* etc.

A diferença entre usar uma classe de associação e um conceito intermediário é sutil. A Figura 6.41 mostra um exemplo de uma reserva sendo modelada como um conceito intermediário e a Figura 6.42 é uma versão dessa reserva modelada como uma classe de associação.

Na caso da Figura 6.41, uma reserva associa um comprador a um livro. No caso da Figura 6.42, o comprador e o livro são associados diretamente, e a reserva, como classe de associação, é consequência da associação. Entretanto, na Figura 6.41, o mesmo comprador pode ter várias reservas para o mesmo livro (nada no modelo impede isso), enquanto na Figura 6.42 um comprador só pode ter uma reserva para cada livro. A ligação entre o comprador e o livro pode existir ou não; a associação não é um *bag*. Entretanto, se ambos os papéis da associação na Figura 6.42 fossem marcados com *{bag}*, então um comprador poderia ter mais do que uma reserva para o mesmo livro e o modelo da Figura 6.42 seria equivalente ao da Figura 6.41.

A questão de se ter diferentes registros da mesma pessoa, como discutido anteriormente, cria, quase sempre, problemas em sistemas de informação. Por exemplo, um estudante entra na universidade e é registrado como tal. Então, ele recebe uma bolsa, e um novo registro dele é criado como bolsista. Então, ele vai trabalhar em um laboratório e é registrado novamente. Finalmente, ele é contratado como professor e um novo registro é criado. Ele aparece várias vezes nos registros da universidade, como se fosse pessoas diferentes. Isso acontece porque sistemas usualmente usam herança (Figura 6.43) ou registros completamente separados (Figura 6.44) nesses casos.

Para evitar este problema, como visto anteriormente, a solução é reconhecer que a pessoa é sempre a mesma. Os papéis que a pessoa representa junto à instituição é que mudam, mas a pessoa não se torna uma nova pessoa. Assim, a solução para esta situação é mais adequada quando for baseada em classes de associação, como mostra a Figura 6.45.

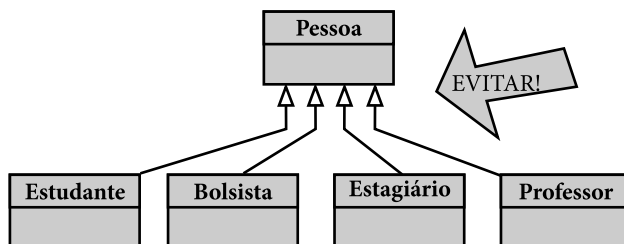


Figura 6.43

Representação inadequada de vários registros para a mesma pessoa com herança.

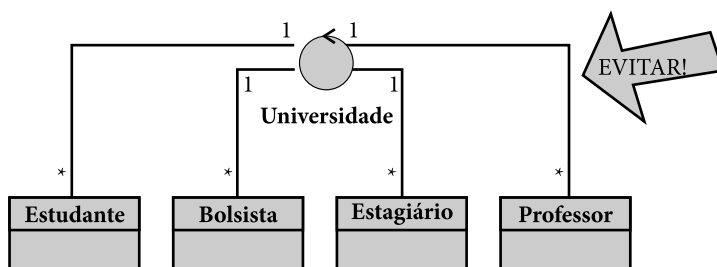


Figura 6.44

Representação inadequada de vários registros para a mesma pessoa como conceitos separados.

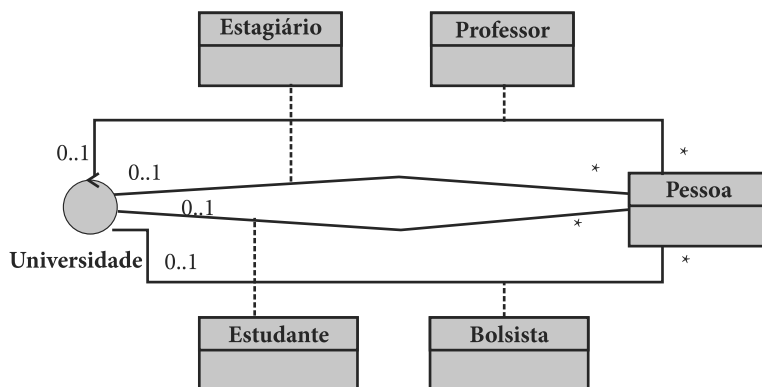


Figura 6.45

Representação de vários registros para a mesma pessoa como classes de associação.

Na Figura 6.45, se uma pessoa puder representar o mesmo papel mais de uma vez (por exemplo, ser estudante em dois cursos na mesma universidade), então a multiplicidade no lado esquerdo da associação deveria ser * também.

Uma exceção razoável poderia ser admitida se apenas um tipo de pessoa existisse no contexto de um sistema. Por exemplo, se o sistema apenas gerencia informação sobre professores e este é o único papel que uma pessoa pode representar na universidade no contexto deste sistema de informação, então seria mais fácil apenas criar uma classe chamada *Professor*, e no contexto daquele sistema, *Professor* seria sinônimo de *Pessoa*, porque não haveria outros tipos de pessoa. O mesmo acontece com o exemplo do sistema *Livir*, no qual foi decidido manter *Comprador* como a única classe que representa pessoas, em vez de transformá-la em uma classe de associação.

6.6.3 Classes modais

Classes modais são usadas para modelar conceitos com instâncias que podem mudar de um estado para outro durante sua existência, modificando, possivelmente, a estrutura de suas propriedades, incluindo atributos, associações e comportamento. Embora algumas linguagens de programação permitam que instâncias troquem sua estrutura pela troca de sua classe, isso não é assumido usualmente como um princípio de modelagem, porque tais mudanças podem criar problemas estruturais imprevisíveis.

Mesmo se uma instância pudesse mudar sua classe, redefinir seus atributos e associações continuaria sendo um problema. Entretanto, existem técnicas de modelagem que permitem que se representem situações que poderiam requerer que objetos trocassem de classe sem usar esta característica. A ideia é não trocar a classe do objeto, mas o seu *estado*. Quando uma TV é ligada, ela não se torna um novo tipo de TV: ela apenas muda seu *estado*.

Identificam-se três situações de complexidade crescente que são relacionadas à modelagem de estados:

- *Transição estável*: os diferentes estados de um objeto não afetam sua estrutura, mas apenas os valores de suas propriedades (atributos e ligações).
- *Transição monotônica crescente*: à medida que o objeto muda de estado, ele pode manter suas propriedades ou adquirir novas.
- *Transição não monotônica*¹⁶: à medida que o objeto muda de estado, ele pode manter, adquirir ou perder propriedades.

Modelar soluções para as transições estável e monotônica crescente é muito simples. Entretanto, para modelar a transição não monotônica, é necessário usar um padrão de design chamado *Estado* (Gamma; Helm; Johnson; Vlissides, 1995), conforme mostrado nas subseções seguintes.

¹⁶ O caso monotônico decrescente também seria considerado aqui (um tipo de transição que pode apenas remover propriedades de uma instância).

6.6.3.1 Transição estável

Frequentemente, diferentes estados de um objeto podem ser representados por um simples atributo. De fato, o conjunto de todos os atributos e links de um objeto define seu estado. Por exemplo, o estado de um pedido poderia ser modelado simplesmente como um atributo chamado *estado* tipado com uma enumeração cujos valores poderiam ser “em andamento”, “concluído” e “pago”.

Outro exemplo é o atributo *suspenso* na classe *Endereço* (Figura 6.46). Se um pacote enviado a um endereço retorna, então o endereço é marcado como suspenso (o atributo é modificado para *verdadeiro*). Quando o comprador atualizar esse endereço, ele será mudado para *falso* novamente. O valor *default* para o atributo *suspenso* é falso.

Se o atributo *suspenso* for verdadeiro, então aquele endereço não pode ser usado para o envio de mercadorias.

A transição é considerada estável porque apenas o valor do atributo muda. A estrutura interna do objeto se mantém a mesma. Isso não acontece com os dois casos apresentados nas subseções seguintes.

6.6.3.2 Transição monotônica crescente

A situação é um pouco mais complexa quando o conceito pode adquirir novos atributos ou associações à medida que ele muda de estado. Por exemplo, faturas que estão aguardando pagamento têm apenas *dataDeVencimento* e *valorDevido* como atributos. Mas uma fatura que já foi paga tem adicionalmente *dataDePagamento* e *valorPago* (Figura 6.47).

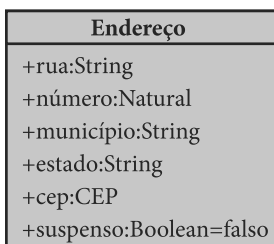


Figura 6.46

Um conceito com transição de estado estável.

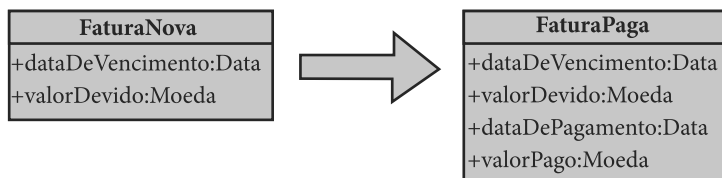


Figura 6.47

Um conceito com transição monotônica crescente.

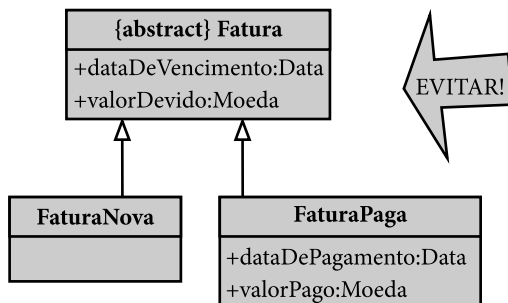


Figura 6.48

Forma inadequada de modelar uma transição monotônica crescente com herança.

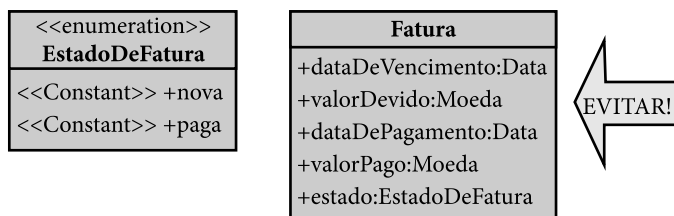


Figura 6.49

Forma inadequada de modelar transição monotônica crescente com uma única classe e atributos opcionais.

Diz-se que a transição da Figura 6.47 é monotônica crescente porque novos atributos e associações podem ser criados para ela, mas não eliminados. Isso implica que uma fatura paga não pode voltar atrás e tornar-se uma fatura nova novamente.

Não seria adequado modelar esta situação com herança, como visto na Figura 6.48, porque nesse caso uma instância de *FaturaNova* deveria ser transformada em uma instância de *FaturaPaga* ou ela teria de ser destruída e criada novamente do zero e ter todas as suas referência reconstruídas.

Outra solução que não seria muito boa, embora seja bem popular ainda, consiste em criar uma única classe *Fatura* e permitir que certos atributos sejam *null* até que a classe mude o seu estado, como mostrado na Figura 6.49. Usualmente, a verificação da consistência da instância é feita nos métodos de atualização. Mas uma *invariante* também poderia ser usada (como explicado na Seção 6.7) para garantir que nenhuma instância alcance um estado inválido, como, por exemplo, tendo *dataDePagamento* definida mas *valorPago* *null*.

Este modelo ainda não é bom, porque ele gera uma classe com baixa coesão, a qual tem, em consequência, regras de consistência complexas que devem ser verificadas frequentemente. Este tipo de classe é altamente susceptível a erros de design e programação.

Seria melhor modelar o conceito de fatura de forma que a consistência dos objetos pudesse ser garantida pela própria estrutura do modelo, e não por regras externas. Como

se trata de transição monotonicamente crescente, então é possível modelar essa situação simplesmente dividindo o conceito original em dois: um que representa a fatura e outro que representa seu pagamento com as propriedades que forem adicionadas quando o pagamento ocorrer. Esses dois conceitos estão ligados por uma associação de 1 para 0..1 (Figura 6.50).

Com o modelo apresentado da Figura 6.50, é impossível que uma fatura tenha data de pagamento definida e valor pago indefinido, e isso é garantido sem adicionar qualquer restrição em seus atributos ou associações.

O estado de uma fatura é um atributo derivado, o qual é calculado como segue: se não há pagamento ligado à fatura, então a fatura é *nova*; caso contrário, ela é *paga*. Em OCL, esta condição pode ser expressa como:

```
Context Fatura:estado
derive:
  if self.pagamento->isEmpty() then
    EstadoDeFatura::nova
  else
    EstadoDeFatura::paga
  endif
```

A expressão anterior introduz duas novas características OCL:

- *if-then-else-endif*, que é uma das funções de seleção em OCL. Se a condição após o *if* é verdadeira, então a função resulta na avaliação da expressão após o *then*; caso contrário, ela resulta na avaliação da expressão após o *else*.
- *isEmpty()*, que é uma função aplicada a uma estrutura de coleção que retorna *true* se a coleção é vazia e *false* caso contrário. Esta é uma forma fácil de verificar se um objeto está ligado a algum outro ou não.

6.6.3.3 Transição não monotônica

Com a *transição não monotônica*, um objeto pode tanto ganhar quanto perder propriedades à medida que ele muda seu estado.

Felizmente, não é muito frequente que um sistema de informação exija que uma informação seja *perdida*. Entretanto, algumas vezes, isso pode ser exatamente o que deve ser feito.

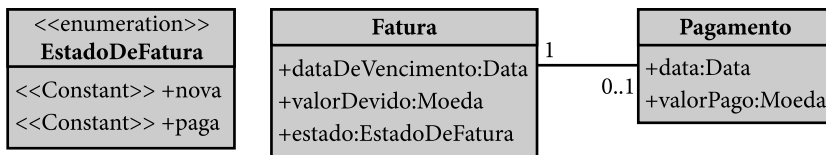
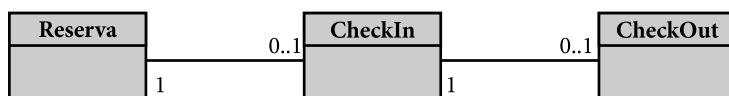


Figura 6.50

Forma efetiva de modelar transição monotônica crescente.


Figura 6.51

Um diagrama de máquina de estado para modelar um processo simples de hospedagem.


Figura 6.52

Modelo possível para reservas como transição monotônica crescente.

Há muitas formas de conceber e modelar um sistema de reservas para um hotel, por exemplo. Uma delas consiste em encarar a hospedagem como uma entidade que evolui a partir de uma reserva em três passos:

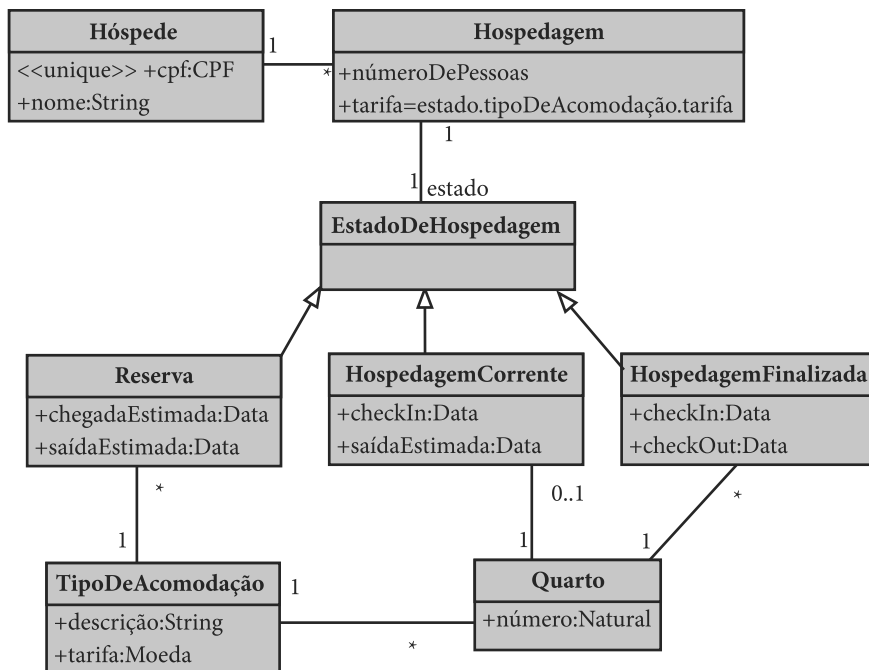
1. Inicialmente, um hóspede potencial faz uma reserva indicando os dias que ele pretende chegar e sair, o tipo e acomodação e o número de pessoas. O hotel fornece a tarifa.
2. Quando o hóspede faz o *check in*, a data de chegada é registrada (ela pode ser o mesmo dia da reserva ou uma data diferente, eventualmente). O hotel atribui um quarto que pode até ser de um tipo diferente daquele que foi inicialmente reservado, e, se esse for o caso, informa ao hóspede a nova tarifa. A data esperada de saída continua a existir, embora ainda possa ser atualizada durante a estada.
3. Quando o hóspede faz *check out*, a data esperada de saída deixa de existir e é registrada a data efetiva de saída, e a conta tem que ser paga.

Este conjunto de estados poderia ser modelado durante a análise de negócio com um diagrama de máquina de estados similar ao da Figura 6.51.

Se uma hospedagem apenas adquire novos atributos e associações à medida que ela evolui por meio dos estados da Figura 6.51, então ela poderia ser representada como uma cadeia de conceitos como a mostrada na Figura 6.52.

Entretanto, é possível assumir que algumas transições de estado possam eliminar alguns atributos ou associações que não tenham mais significado no novo estado, como, por exemplo, datas estimadas, as quais podem ser trocadas pelas datas reais, e o tipo de quarto, que pode ser trocado por um quarto real, o qual já tem um tipo.

Para representar este tipo de situação, é necessário usar um padrão de design conhecido como *Estado* (Gamma; Helm; Johnson; Vlissides, 1995). De acordo com este padrão, uma classe deveria ser separada de seus estados. Propriedades que são compartilhadas por todos os estados são mantidas na classe original, e propriedades que pertencem apenas a um ou a poucos estados são movidas para seus respectivos estados. Esses estados são especializações de uma classe abstrata, como mostrado na Figura 6.53.

**Figura 6.53**

Transição não monotônica modelada com o padrão de design *Estado*.

Na Figura 6.53, os atributos *númeroDePessoas* e *tarifa* existem em todos os estados possíveis de uma hospedagem e é por isso que eles são representados na classe *Hospedagem*. As outras propriedades estão distribuídas entre os três possíveis estados:

- *Reserva*: uma hospedagem, em adição ao número de pessoas, tem as datas de chegada e saída estimadas e o tipo de acomodação (ainda não um quarto específico), a partir dos quais o valor da tarifa da hospedagem é obtido.
- *HospedagemCorrente*: uma hospedagem, em adição ao número de pessoas e tarifa tem uma data de *check in* efetiva, data de saída estimada e um quarto efetivamente atribuído. O atributo *chegadaEstimada* e o link direto para o tipo de acomodação não são mais necessários e podem ser descartados.
- *HospedagemFinalizada*: uma hospedagem, em adição ao número de pessoas e tarifa, tem datas efetivas de *check in* e *check out* e um quarto efetivamente atribuído. A saída estimada não é mais necessária e pode ser descartada.

Do ponto de vista de um *Quarto*, ele pode estar ligado a, no máximo, uma *HospedagemCorrente* (0..1), mas pode estar associado a várias *HospedagemFinalizada* (*). Assim, a propriedade temporal dessa associação já está modelada também, porque a associação de um quarto com uma hospedagem pode ser de, no máximo, uma no presente, mas muitas no passado.

6.7 Invariantes

Existem situações nas quais a expressividade dos diagramas não é suficiente para representar regras que poderiam ser registradas no modelo conceitual. Nesses casos, o analista deveria usar *invariantes*.

Invariantes são restrições gerais sobre instâncias das classes. Algumas restrições são representadas nos papéis das associações; por exemplo, uma restrição que estabelece que um pedido não deveria ter mais do que cinco itens poderia ser representada como na Figura 6.54.

Entretanto, nem toda restrição pode ser representada nos papéis das associações. Considere o modelo mostrado na Figura 6.55.

O *valorTotal* de um pedido é um atributo derivado na Figura 6.55. A função *soma* em OCL retorna o somatório dos elementos numéricos que são produzidos para cada um dos elementos do conjunto. A expressão entre parênteses após o *sum* indica como cada valor numérico individual é obtido a partir dos elementos com conjuntos. A expressão usada para definir o *valorTotal* na Figura 6.55 é uma abreviação de:

```
Context Pedido::valorTotal:Moeda
derive:
    self.item->sum(umItem|umItem.subtotal)
```

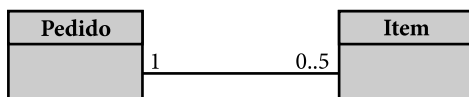


Figura 6.54

Uma restrição que pode ser representada como uma multiplicidade de papel.

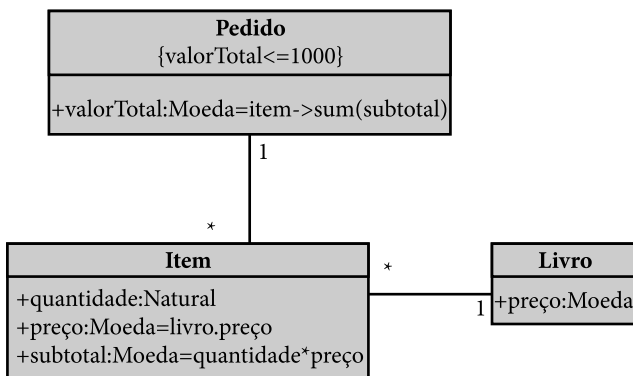


Figura 6.55

Um exemplo de uma classe com uma invariante.

Alguém poderia também tentar escrever a expressão acima como *self.item.subtotal->sum(umSubtotal|umSubtotal)* ou *self.item.subtotal->sum()*. Entretanto, nesse caso, como o papel *item* é um conjunto e não um *bag* ou sequência, os subtotais também seriam um conjunto. Se dois itens tiverem o mesmo valor como subtotal, então ele poderia aparecer apenas uma vez no conjunto e a soma não refletiria as intenções do modelador.

O subtotal na classe *Item* é também um atributo derivado calculado pela seguinte expressão:

```
Context Item::subtotal:Moeda
  derive:
    self.quantidade*self.preço
```

O preço do item tem um valor inicial definido como o preço do livro que foi ligado ao item:

```
Context Item::preço:Moeda
  init:
    self.livro.preço
```

Agora, se houver uma restrição que estabelece que nenhum pedido pode ter um valor total maior do que R\$ 1.000,00, isso não poderia ser representado na multiplicidade de papel, como feito na Figura 6.54. Entretanto, a restrição pode ser representada pelo uso de uma *invariante* como a seguinte:

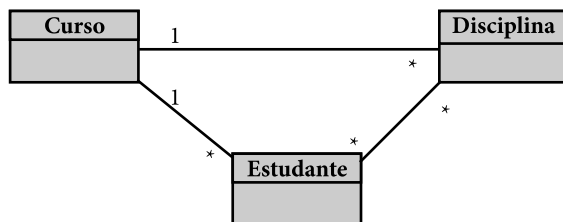
```
Context Pedido
  inv:
    self.valorTotal<=1000
```

Invariantes podem aparecer no diagrama indicadas como restrições associadas a uma classe. A invariante mencionada anteriormente é representada logo abaixo do nome da classe *Pedido* na Figura 6.55.

A declaração de invariante em OCL não tem subcontexto; ela deve ser verdadeira para qualquer instância da classe, não para um atributo ou papel de associação particular.

Muitos desenvolvedores de software, quando se defrontam com a necessidade de usar invariantes, escolhem incorporar essas regras nos métodos que atualizam os atributos e ligações dos objetos. Por exemplo, considerando a Figura 6.55, um teste para a restrição deveria ser adicionado ao método que adiciona um novo item ao pedido para verificar se o valor total ultrapassa R\$ 1.000,00; nesse caso, a operação deveria ser bloqueada.

O problema com essa abordagem é que a verificação acabaria sendo feita apenas *naquele* método, e então ela não seria uma regra geral para todo e qualquer método. O desenvolvedor conhece a regra hoje, mas o que acontece se outro desenvolvedor assumir o projeto mais tarde? E durante a manutenção do sistema? Um desenvolvedor que fizer mudança no sistema alguns anos depois poderá não saber nada sobre essa regra. Ele provavelmente não revisaria o documento de requisitos (assumindo que ele ainda exista e esteja atualizado), e poderia potencialmente introduzir erros conceituais no sistema, se ele permitisse que outros métodos violassem essa regra.


Figura 6.56

Um modelo que necessita de uma restrição adicional para ser consistente.

Por exemplo, se a verificação for implementada apenas na operação que adiciona um novo item a um pedido, o que acontece se o preço do item for alterado depois disso? O que acontece se a quantidade do item for alterada depois que ele é inserido no pedido?

Assim, *regras conceituais gerais que sempre se aplicam devem estar explícitas no modelo conceitual, e não relegadas a verificações no código de programação*. Instâncias inconsistentes podem ser evitadas; se for possível, tais restrições devem ser representadas na estrutura do diagrama; caso contrário, elas devem ser representadas por invariantes.

Outro exemplo, que acontece com frequência, é a necessidade de restringir duas associações de forma que uma dependa da outra. Na Figura 6.56, note que estudantes estão associados a um curso, cada curso tem disciplinas e estudantes se matriculam em disciplinas.

Entretanto, o modelo representado na Figura 6.56 não garante que estudantes escolherão disciplinas associadas ao seu curso. De acordo com o modelo, um estudante poderá se matricular em qualquer disciplina, e se este não for o caso, uma restrição deveria ser adicionada ao modelo (com o uso de linhas tracejadas). Esta notação é uma forma alternativa para uma invariante.

Para garantir que um estudante só possa se matricular em disciplinas relacionadas ao seu próprio curso, uma invariante como a seguinte deveria ser fornecida:

```

Context Estudante
inv:
    if self.disciplina->notEmpty() then
        self.disciplina.curso=self.curso
    endif

```

A expressão usa uma característica de conjuntos (não permitir que elementos sejam repetidos) para verificar que todas as disciplinas cursadas pelo estudante sejam de seu curso. A expressão *self.curso* resulta no curso do estudante (um conjunto que só pode conter um único elemento em razão da multiplicidade deste papel). No entanto, a expressão *self.disciplina.curso* resulta no conjunto com todos os cursos associados a todas as disciplinas cursadas pelo estudante. Se houverem disciplinas de cursos diferentes, então este conjunto não poderá ser igual a *self.curso* (o qual tem um único elemento). Caso contrário, se todos os cursos são o mesmo, então o conjunto *self.disciplina.curso* vai conter um único elemento

(porque conjuntos não repetem elementos), se este elemento for o mesmo de *self.curso*, então a invariante será verdadeira e válida.

6.8 Construção iterativa do modelo conceitual

Esta seção mostra como o modelo conceitual pode ser refinado com informação obtida nos casos de uso expandidos. Inicialmente, discutimos como encontrar conceitos e atributos nos textos dos casos de uso. Em seguida, discutimos como identificar associações no modelo. Finalmente, apresentamos um exemplo completo com alguns casos de uso das primeiras iterações do projeto Livir e sua contribuição para o modelo conceitual.

6.8.1 Como encontrar conceitos e atributos

O processo de descobrir os elementos de um modelo conceitual podem variar. Entretanto, uma das técnicas mais úteis é olhar para o texto dos casos de uso expandidos ou diagramas de sequência de sistema. A partir desses artefatos, pode-se descobrir todos os elementos textuais que eventualmente se referem a informação que deve ser gerenciada.

Usualmente, esses elementos textuais são compostos de *substantivos* tais como “pessoa”, “pedido”, “pagamento” etc., ou por expressões que denotam substantivos (conhecidas em linguística como *sintagmas nominais*), tais como “autorização de pagamento”. Além disso, algumas vezes, mesmo verbos podem representar conceitos, quando verbos expressam um ato que corresponde a um substantivo, como, por exemplo, o verbo “pagar”, que corresponde ao substantivo “pagamento”.

O analista deve ter em mente as metas do projeto e o escopo quando estiver olhando para os elementos do modelo conceitual. Não é recomendável representar no modelo conceitual informação que é irrelevante para o sistema. Assim, nem todo substantivo, adjetivo ou verbo será considerado um elemento do modelo. O analista é responsável por entender quais são as reais necessidades de informação do sistema e filtrar as irrelevâncias.

O processo de identificação de conceitos e atributos consiste no seguinte:

1. Identificar no texto do caso de uso expandido ou nos argumentos das operações do diagrama de sequência de sistema as palavras ou expressões que correspondem a conceitos que são relevantes para se manter registro.
2. Agrupar palavras ou expressões que se referem à mesma entidade como, por exemplo, “compra” e “aquisição”, “comprador” e “cliente” etc.
3. Decidir quais dos elementos identificados são conceitos complexos e quais são meros atributos. Usualmente, atributos são os elementos que podem ser considerados alfanuméricos (nomes, números, códigos, valores monetários, booleanos etc.) enumerações ou primitivos (datas, ISBNs etc.).

Aplicando esta técnica ao caso de uso da Figura 5.10, os principais elementos relacionados ao caso de uso podem ser encontrados. A Figura 6.57 mostra estes elementos sublinhados.

Caso de uso 01: *Pedir livros*

1. [IN] O comprador fornece palavras-chave para pesquisar livros.
2. [OUT] O sistema gera uma lista de livros para venda que satisfaz as palavras-chave incluindo pelo menos título, autor, preço, número de páginas, editora, ISBN e imagem de capa.
3. [IN] O comprador seleciona livros da lista e indica a quantidade desejada para cada um.
4. [OUT] O sistema gera um resumo do pedido (título, autor, quantidade, preço unitário e subtotal para cada livro) e o valor total.
5. [IN] O comprador finaliza o pedido.

Exceção 3a: A quantidade solicitada é maior do que a quantidade em estoque para um ou mais livros.

- 3a.1. [OUT] O sistema informa as quantidades disponíveis para cada livro que foi pedido acima do estoque.
- 3a.2. [IN] O comprador altera as quantidades desejadas para valores iguais ou menores do que o disponível no estoque.
- 3a.3. Avança para o passo 4.

Exceção 5a: O comprador ainda não se identificou.

- 5a.1. [IN] O comprador fornece uma identificação válida.
- 5a.2. Retorna ao passo 5.

Exceção 5b: O comprador não tem uma conta.

- 5b.1. [IN] O comprador se registra e fornece nome de usuário, senha e endereço de email.
- 5b.2. Retorna ao passo 5.

Exceção 5c: Nenhum livro foi selecionado para compra.

- 5c.1. Retorna ao passo 1.

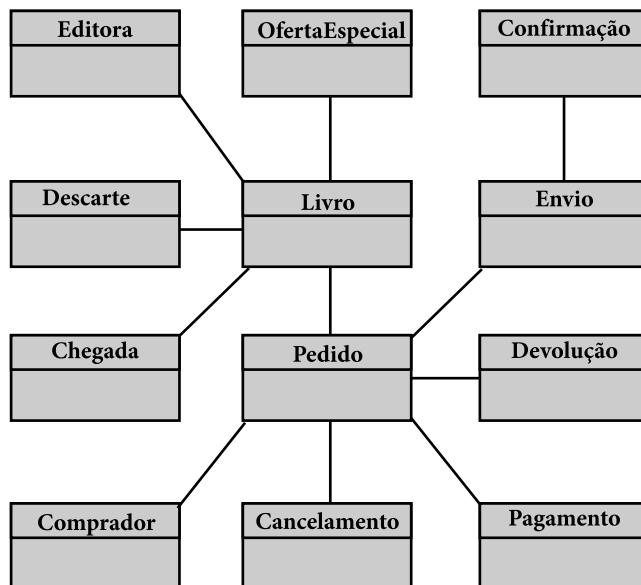
Variante 5d: O usuário deseja pesquisar mais livros.

- 5d.1. Retorna ao passo 1.
-

Figura 6.57

Caso de uso com conceitos candidatos e atributos sublinhados.

Mesmo se não houver modelo conceitual preliminar, este caso de uso é suficiente para identificar alguns conceitos que pertencem ao domínio do sistema. Se o modelo conceitual preliminar já existe, então o caso de uso ajuda a refinar o modelo, indicando novos conceitos, novos atributos ou mudanças estruturais. Para o exemplo, assumimos que já existe um modelo conceitual preliminar, o qual é mostrado da Figura 6.58.

**Figura 6.58**

Modelo conceitual preliminar para o exemplo corrente.

Os termos identificados no texto do caso de uso da Figura 6.57 são, em ordem de aparecimento:

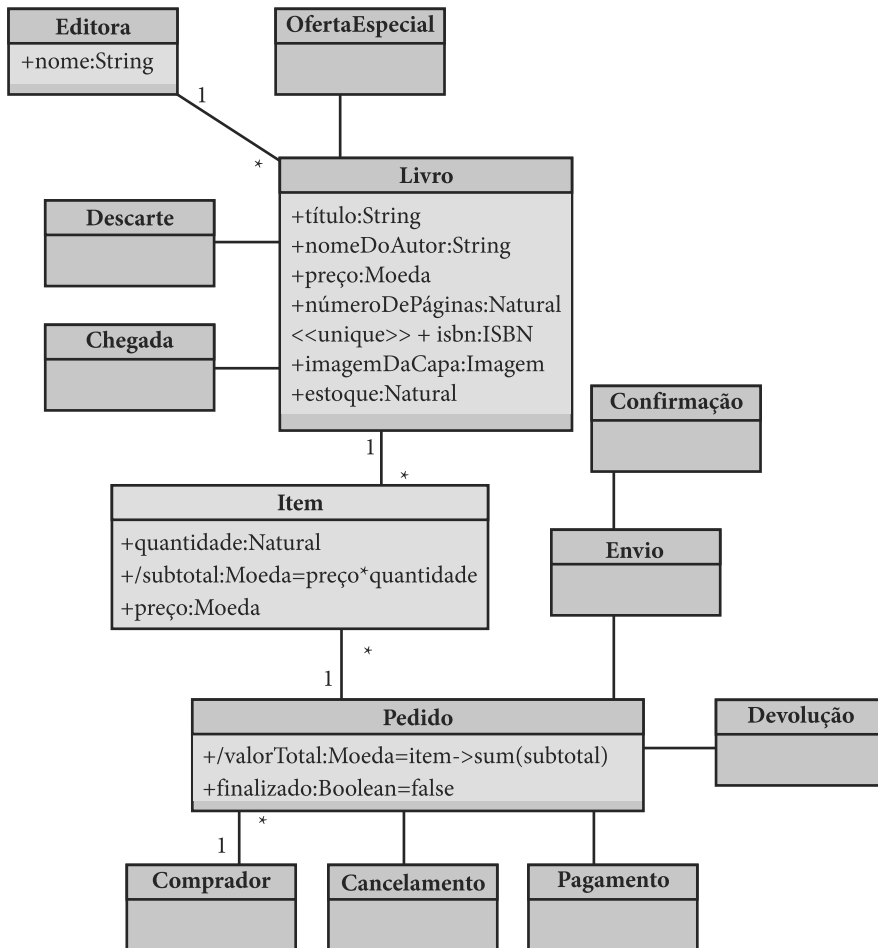
- *Comprador*: uma classe que já está no modelo conceitual.
- *Palavra-chave*: palavras-chave são usadas para pesquisa; elas não são consideradas neste ponto como atributos de conceitos complexos (embora elas possam vir a ser, caso um histórico ou preferências de pesquisa venham a ser registrados).
- *Livro*: uma classe que já está no modelo conceitual.
- *Venda*: no contexto desse caso de uso e nesse ponto da análise “venda” e “pedido” podem ser considerados sinônimos. Assim, esta é uma classe que já aparece no modelo conceitual.
- *Título, autor, preço e número de páginas*: atributos de *Livro*. *Autor* poderia ainda ser considerado um conceito complexo se a informação sobre autores fosse mais complexa do que um simples nome.
- *Editora*: uma classe que já está no modelo. O caso de uso provavelmente está se referindo ao nome da editora.
- *ISBN e imagem da capa*: atributos de *Livro*.
- *Quantidade desejada*: não pode ser considerada um atributo de *Livro* porque ela varia dependendo do pedido. Ela não pode ser considerada atributo do pedido, porque varia dependendo do livro (pode haver vários livros em um pedido). A quantidade desejada é, portanto, um atributo da associação entre um livro e um pedido, representada por um conceito intermediário ou por uma classe de associação denominada *Item*.

-
- *Resumo do pedido*: isso se refere a um *Pedido*; não parece ser um novo conceito.
- *Título, autor* (ou *nome do autor*), *quantidade*, *preço unitário* e *subtotal*: exceto por *subtotal*, todos esses atributos já foram mencionados. *Subtotal* pode ser um atributo derivado de *Item*. *Preço unitário* é equivalente a *preço*, o qual já é considerado um atributo do *Livro*. Entretanto, ele pode de fato ser também um novo atributo para a classe *Item*.
- *Valor total*: atributo derivado de *Pedido*.
- *Finaliza o pedido*: isso indica uma ação que representa uma mudança de estado em um pedido. Talvez um atributo seja suficiente para representar este estado, porém mais análise poderia ser recomendável para verificar se a finalização de um pedido não é uma operação mais complexa (o que de fato ela é).
- *Quantidade solicitada*: é a mesma coisa que *quantidade*.
- *Quantidade em estoque*: provavelmente um novo atributo de *Livro*.
- *Quantidades disponíveis*: sinônimo de *quantidade em estoque*.
- *Identificou*: seria um estado persistente do comprador? Talvez não. Ele se refere a se o comprador fez o login ou não.
- *Identificação válida*: isso pertence ao domínio da segurança. Entretanto, o modelo conceitual provavelmente deveria ter um meio de identificar compradores. Usualmente, números de identificação como o CPF são usados para este propósito.
- *Conta*: isso novamente tem a ver com login.
- *Nome de usuário, senha e endereço de email*: dependendo do tipo de *framework* de segurança que estiver sendo usado (ou não), estes podem ser atributos de um comprador (ou atributos de um *usuário* que esteja ligado a um comprador), ou ainda eles podem ser completamente tratados pelo *framework* de segurança e ignorados no modelo conceitual. Por enquanto, eles são deixados como tópicos de segurança e não incluídos no modelo conceitual.

Da informação coletada anteriormente, o modelo conceitual preliminar pode ser refinado, e ele poderia se tornar similar ao da Figura 6.59, no qual novas classes e atributos foram destacados.

Note que um único caso de uso adicionou nova informação significativa ao modelo. Isso não é coincidência, porque este caso de uso foi considerado o mais importante no sistema e assim o primeiro a ser detalhado nas iterações. Sua complexidade tem muito a contribuir para a compreensão da estrutura da informação que o sistema vai gerenciar.

Nem toda associação da Figura 6.59 tem suas multiplicidades definidas. O *default* da UML para papéis de associação com multiplicidade indefinida é 1, mas a situação da Figura 6.59 é de que associações com multiplicidade indefinida simplesmente ainda não foram analisadas a este ponto e são consideradas como “a serem definidas”.

**Figura 6.59**

Segunda versão do modelo conceitual, refinado após a análise do caso de uso 01: *Pedir livros*.

6.8.2 Conceitos dependentes e independentes

Um conceito é *dependente* de outro e, para fazer sentido, suas instâncias devem estar ligadas a instâncias do outro conceito, ou seja, a instância isolada não representa informação que seja minimamente significativa, no domínio considerado¹⁷. Olhando para a Figura 6.59, pode-se notar que um pedido não pode fazer sentido a não ser que esteja associado a um comprador e uma lista de itens. Sem essas associações, o conceito de *Pedido* não faz sentido. O pedido também tem outras associações na figura, mas elas são opcionais para ele.

¹⁷ Existe um paralelo entre os conceitos dependentes e os verbos transitivos. A sentença “ele abriu” não faz sentido, porque o verbo necessita de um complemento, como, por exemplo, “ele abriu a lata”.

Um conceito é *independente* se ele tem significado mesmo sem estar associado a outros¹⁸. Por exemplo, o conceito *Editora* pode ser entendido apenas a partir de seus atributos, sem a necessidade de estar associado a outros conceitos. As associações existentes para uma editora são opcionais. Assim, o conceito *Editora*, neste sentido, é independente.

Mas a que propósito serve essa distinção? Foi discutido no Capítulo 5 que apenas os conceitos mais simples podem ser gerenciados por casos de uso CRUD. Conceitos dependentes não devem ser considerados simples a princípio, enquanto os independentes podem ser simples em princípio.

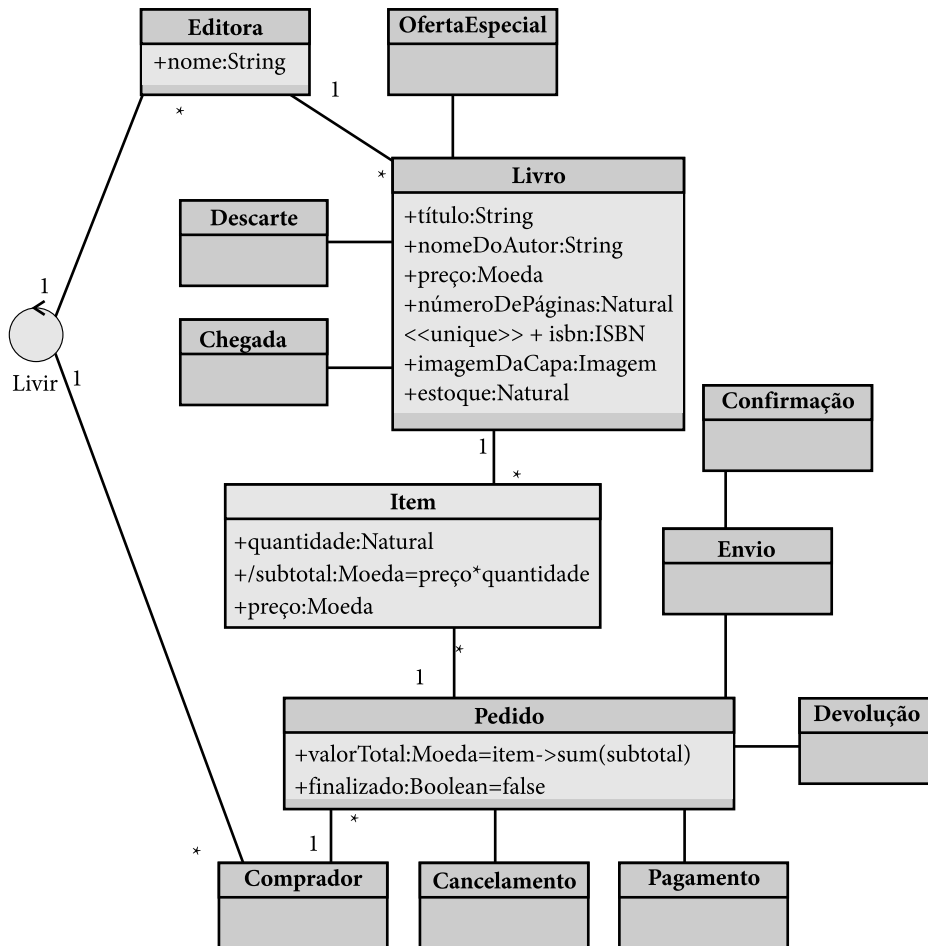
Na Figura 6.59, quase todos os conceitos são dependentes. Por exemplo, um envio depende de um pedido, uma confirmação depende de um envio, e assim por diante. Apenas *Editora*, *Livro* e *Comprador* são conceitos independentes e candidatos a serem gerenciados por casos de uso CRUD.

Conceitos que são gerenciados por casos de uso CRUD são usualmente aqueles que podem ser acessados diretamente quando a informação sobre eles é recuperada de um sistema. Embora a adição do controlador-fachada ao modelo pudesse aguardar pelas atividades de design (Capítulo 9), isso pode ser feito agora para indicar quais conceitos são independentes. Em princípio, apenas os conceitos independentes têm ligações obrigatórias com a classe controladora. Estar ligado ao controlador significa que as instâncias da classe podem ser acessadas diretamente. Por exemplo, uma instância de *Livro* pode ser acessada diretamente pelo seu ISBN. Entretanto, uma instância de *Item* não é acessível diretamente. Para acessar um *Item*, é necessário procurar a partir de um *Livro* ou de um *Pedido*. Assim, a classe *Item* deve ser ligada ao controlador fachada.

Embora *Livro* seja um conceito independente, ele não foi ligado ao controlador, porque ele já tem uma associação obrigatória com outro conceito independente: a *Editora*. Nesse caso, *qualquer* livro pode ser acessado por uma pesquisa no conjunto de livros das editoras. Se mais tarde for descoberto que livros devem ser acessados diretamente, independentemente de sua editora, ou que qualquer outro conceito necessite deste tipo de acesso, então associações (derivadas) para o controlador podem ser criadas na medida do necessário, mas elas não são criadas por *default* para qualquer conceito, para evitar aumentar o acoplamento entre as classes.

Existem, portanto, dois caminhos principais para alcançar a informação no modelo da Figura 6.60: a partir de uma editora ou a partir de um comprador. Todos os outros conceitos são alcançados por um caminho que inicia em um desses dois conceitos. Mais tarde, como mencionado, necessidades de informação poderão exigir a definição de outros caminhos. No Capítulo 9, vemos que um pedido também pode ser acessado diretamente porque certo número de operações de sistema necessitam obter um pedido a partir de seu número, independentemente de quem seja o comprador ou quais sejam os livros.

¹⁸ Existe um paralelo entre os conceitos independentes e os verbos intransitivos. A sentença “ele dormiu” faz sentido porque o verbo não exige complemento. Entretanto, mesmo verbos intransitivos podem ter informação suplementar (por exemplo, “ele dormiu por três horas”), assim como os conceitos independentes.

**Figura 6.60**

Modelo conceitual refinado com um controlador fachada.

6.8.3 Como encontrar associações

Se a informação relacionada a conceitos e atributos é facilmente encontrada nos textos dos casos de uso, as associações não o são. Como mencionado antes, casos de uso indicam muitas operações que transformam informação, mas deve-se procurar neles as pistas que levam às associações.

Então, como podemos encontrar associações entre conceitos complexos? Existem pelo menos duas regras que podem ser usadas:

- *Conceitos dependentes* (como *Pedido*, *Pagamento*, *Cancelamento* etc.) devem estar necessariamente associados aos conceitos que os complementam (os quais podem ser dependentes ou independentes).

- *Informação associativa*, ou seja, associações que não são obrigatórias, mas que complementam as informações no modelo (tais como “uma pessoa possui um carro”), podem ser representadas por associações.

Algumas vezes, informações associativas aparecem no texto dos casos de uso como conceitos. Por exemplo, um pedido poderia ser uma associação entre um comprador e livros, mas ela seria suficientemente complexa para ser representada como um conceito. Como um pedido é um conceito dependente, existem associações entre esses conceitos e seus complementos.

Quando alguém estabelece que uma pessoa pode possuir um carro, essa informação deveria ser representada por uma associação. Muitos profissionais que trabalham com programação, mas não com modelagem, poderiam produzir modelos similares ao apresentado na Figura 6.61. Entretanto, isso é inadequado, porque (primeiro) atributos não deveriam ser tipados como conceitos, e (segundo) o modelo está disfarçando uma associação real.

Também é inadequado usar chaves estrangeiras (Date, 1982), como adiante na Figura 6.62, porque isso também disfarça associações. Esse erro é comumente cometido por modeladores acostumados a projetar bancos de dados relacionais. Chaves estrangeiras não devem ser usadas em modelagem conceitual porque elas são uma técnica de *implementação*.

Assim, quando dois conceitos complexos estão relacionados um ao outro, a construção a ser usada é a associação (Figura 6.63).

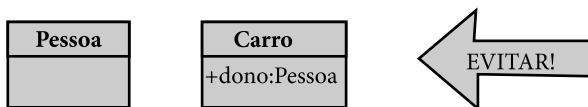


Figura 6.61

Um atributo disfarçando uma associação.

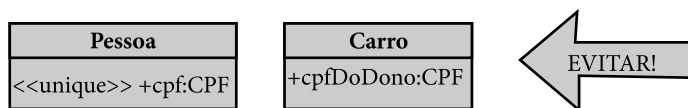


Figura 6.62

Uma associação disfarçada por um atributo como chave estrangeira.

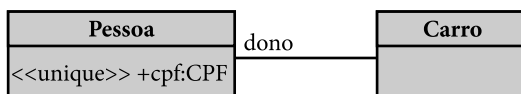


Figura 6.63

Representação adequada para a situação mostrada nas Figuras 6.61 e 6.62.

O exemplo da Figura 6.63, ao contrário dos anteriores, respeita a regra de *coesão* que estabelece que um conceito só pode conter *suas próprias propriedades*. Como o CPF é um atributo do dono, ele não deve ser representado como um atributo de um carro (como foi feito na Figura 6.62). Outra razão para ter associações visíveis e não disfarçadas é mais prática: é mais fácil ver quais objetos têm referências para quais outros. Mais adiante (Capítulo 9), discutiremos como essas linhas de *visibilidade* podem ser usadas para projetar código orientado a objetos efetivo e de boa qualidade.

6.8.4 Exemplo de construção iterativa do modelo conceitual

Foi mencionado anteriormente que o Processo Unificado é dirigido por casos de uso e centrado na arquitetura. Mas o que isso significa na prática? Já vimos como identificar e detalhar casos de uso e também vimos que eles podem ser usados como uma fonte e requisitos para refinar o modelo conceitual, o qual é a base para o diagrama de classes de design, uma importante parte da arquitetura do sistema. A Seção 6.8.1 mostra como o modelo conceitual preliminar poderia ser refinado com informação do primeiro caso de uso detalhado (*Pedir livros*).

Aplicando este processo novamente, simulando-se uma segunda iteração para o caso de uso 02 *Pagar pedido*, novos conceitos e atributos são descobertos (Figura 6.64).

Caso de uso 02: *Pagar pedido*

1. O sistema gera o resumo do pedido pendente (título, autor, quantidade, preço unitário e subtotal para cada livro), bem como o valor total do pedido e uma lista dos endereços registrados para o comprador.
 2. O comprador seleciona um endereço de entrega.
 3. O sistema apresenta a taxa de entrega, a data de chegada prevista e uma lista de cartões de crédito já registrados para o comprador (bandeira e últimos quatro dígitos do número do cartão de crédito).
 4. O comprador seleciona um dos seus cartões para pagamento.
 4. O sistema envia para a respectiva operadora de cartão de crédito os seguintes dados: número do cartão, nome do titular, validade, código de segurança, valor total da compra e código da loja.
 6. A operadora de cartão de crédito aprova a venda pelo envio de um código de autorização.
 7. O sistema informa ao comprador que a compra foi aprovada e apresenta o número de registro para rastreamento do pacote.
-

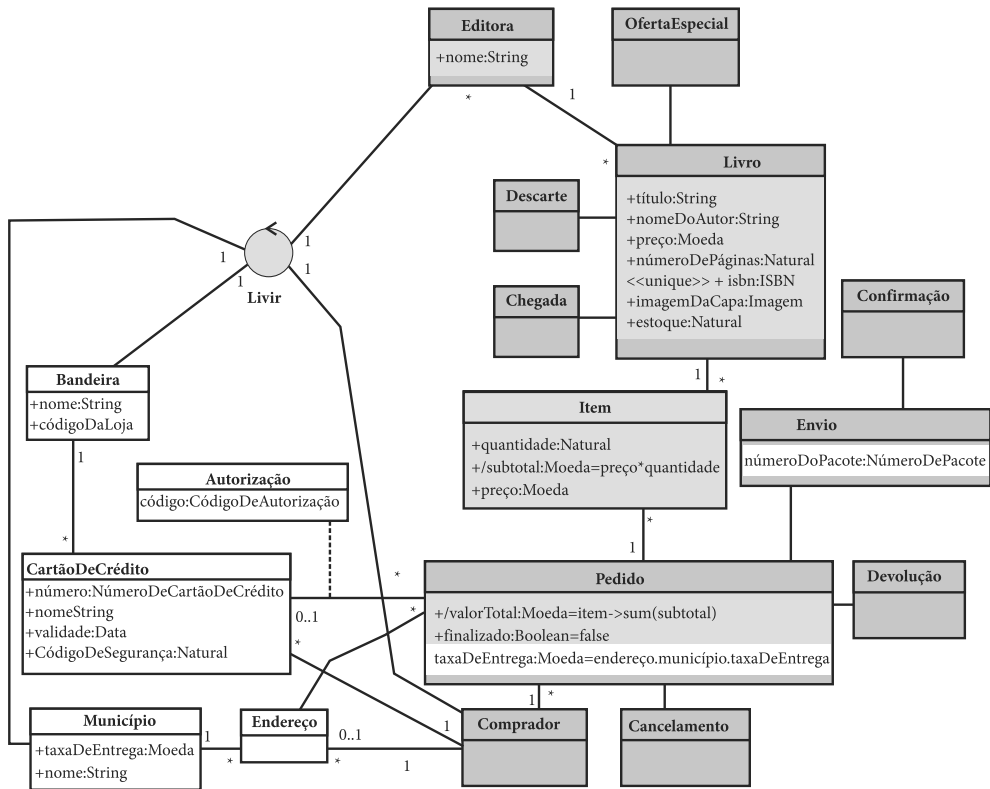
Figura 6.64

Fluxo principal do caso de uso 02 *Pagar pedido* com conceitos e atributos candidatos identificados.

Os elementos identificados nesse caso de uso são:

- *Endereços registrados*: já que um comprador pode ter mais do que um endereço para entrega e como endereços são usados para calcular taxas de entrega, eles não podem ser representados como um simples atributo, mas, em vez disso, devem ser representados como uma classe.
- *Endereço de entrega*: refere-se ao endereço específico associado ao pedido corrente. Assim, uma nova associação é encontrada entre um pedido e um endereço.
- *Taxa de entrega*: atributo do pedido. Mas de onde este valor é obtido? Se os requisitos forem mais aprofundados, descobre-se que diferentes cidades têm taxas distintas que dependem fundamentalmente da distância da cidade até os depósitos da empresa. Assim, uma nova classe é criada (*Município*), a qual é associada ao *Endereço* e uma tarifa *default* para cada cidade é definida como atributo daquela classe.
- *Cartões de crédito*: conceito novo. Deveria ser associado ao comprador.
- *Bandeira*: provavelmente não é um atributo, mas um conceito complexo associado ao cartão de crédito.
- *Cartões para pagamento*: uma associação entre um pedido e um cartão de crédito do seu comprador.
- *Operadora de cartão de crédito*: este é, de fato, um ator externo; seus dados podem ser associados à bandeira do cartão de crédito.
- *Número do cartão, nome do titular, validade e código de segurança*: atributos do cartão de crédito.
- *Valor total da compra*: é o mesmo que *valor total do pedido*.
- *Código da loja*: não é um atributo da livraria, como se poderia pensar. Este código identifica a livraria para cada operadora de cartão de crédito. Como diferentes operadoras podem fornecer diferentes códigos, este atributo deveria ser representado no conceito referente à bandeira do cartão de crédito, a não ser que uma rede de livrarias esteja sendo modelada.
- *Aprova*: isso acontece quando um código de autorização é enviado (ver próximo item).
- *Código de autorização*: pode ser pensado como um atributo do pedido. Mas ele pode apenas existir quando um cartão de crédito já está ligado ao pedido. Assim, ele seria mais bem colocado como um atributo da associação entre o cartão de crédito e o pedido.
- *Número de registro para rastreamento do pacote*: pode ser um atributo do envio.

Os novos elementos estão destacados na Figura 6.65, a qual é a terceira versão do modelo conceitual do exemplo Livir. Note que ainda há conceitos sem atributos e algumas associações foram deixadas com multiplicidade indefinida. Isso ocorre porque os casos de uso analisados até este ponto não produziram informação para refinar esses elementos. Eles poderão ser refinados mais tarde, quando outros casos de uso forem expandidos e sua informação incorporada no modelo conceitual.

**Figura 6.65**

Modelo conceitual refinado com informação do fluxo principal do caso de uso da segunda iteração: 02 *Pagar pedido*.

Observe que apenas o *fluxo principal* deste caso de uso foi usado no exemplo. Fluxos alternativos desse caso de uso ainda precisariam ser analisados e incluídos. Recentemente, Jacobson *et al.* (2011) propuseram a tecnologia de *caso de uso* 2.0, a qual especificamente sugere que um caso de uso deva ser abordado em *fatias*, e não como um todo indivisível.

Pelo menos duas invariantes ainda poderiam ser adicionadas ao diagrama da Figura 6.65 para evitar a existência de informação inconsistente: o endereço do pedido deve ser um dos endereços do comprador do pedido e o cartão de crédito usado para pagar o pedido deve ser um dos cartões de crédito do comprador. Essas invariantes podem ser definidas na classe *Pedido* como:

Context Pedido

inv:

```
self.comprador.endereço->includes(self.endereço) and
self.comprador.cartãoDeCrédito->includes(self.cartãoDeCrédito)
```

O predicado OCL *includes* verifica se o objeto recebido como argumento está incluído no conjunto. Ele é equivalente ao operador matemático “ $\supseteq$ ”. Por exemplo, *self.comprador.endereço* $\rightarrow$ *includes*(*self.endereço*) significa *self.comprador.endereço* $\supseteq$ *self.endereço*. Assume-

-se que se o elemento recebido como argumento for o conjunto vazio, então a expressão é verdadeira. Assim, se o endereço de entrega ou cartão de crédito ainda não foram definidos, a invariante não está sendo violada.

A análise do segundo caso de uso incrementou o modelo conceitual com um número significativo de novas classes e atributos. A ideia é que a quantidade de novidades vá diminuindo à medida que as iterações avançam, porque no final apenas os CRUDs de elementos já estudados e relatórios com informação que já deveria estar no sistema serão tratados. Não se espera que eles venham a trazer informação nova para o modelo conceitual.

Observe que inverter essa ordem poderia fazer com que as iterações finais tivessem a inserção de muitas classes e atributos, e isso iria requerer muito retrabalho no design e algumas refatorações da arquitetura seriam provavelmente necessárias para acomodar essas necessidades. É por isso que é tão fundamental iniciar o projeto pelos casos de uso mais complexos e apenas depois incorporar os mais simples na arquitetura.

6.9 O processo visto aqui

	Concepção	Elaboração
Modelagem de negócio		
Requisitos		
Análise e design		Refinar o modelo conceitual: • Identificar conceitos, atributos e associações no texto dos casos de uso expandidos. • Detalhar atributos e associações com estereótipos, multiplicidade e restrições na medida do necessário. • Organizar o modelo com o uso de herança, classes de associação e especificações temporais. • Adicionar invariantes na medida do necessário.
Implementação		
Teste		
Gerenciamento de Projeto		

6.10 Questões

1. Tente imaginar uma situação do mundo real na qual a estrutura correta de modelagem seja a *sequência*. Lembre-se de que uma sequência verdadeira permite que elementos se repitam e atribui posição aos elementos. Tente não ser enganado pelo uso da palavra “lista” na linguagem natural. Uma “lista de compras”, por exemplo, não é uma sequência porque os elementos não se repetem e sua posição na lista é irrelevante; uma lista de chamada de estudantes pode ser organizada em ordem alfabética, mas, novamente, a repetição não é permitida e a ordem é novamente irrelevante, embora conveniente. Assim, essas não são sequências verdadeiras.
2. Qual o propósito da classe controladora de sistema? Qual é sua relação com conceitos dependentes e independentes?
3. Por que um atributo não deveria ser tipado com um nome de classe? Por que um atributo não deveria ser usado para referenciar outra classe?

4. Elabore uma lista de conceitos e subconceitos (subtipos do conceito original, como, por exemplo, *cão* e *beagle*) do senso comum. Tente então identificar o tipo de cada relação: estrutural, associativa ou temporal. Tente ter pelo menos um exemplo de cada tipo em sua lista.
5. Qual a diferença entre uma associação normal, uma agregação e uma composição?
6. Liste alguns exemplos de tipos de dados que poderiam ser definidos como primitivos.

Modelagem conceitual: padrões

TÓPICOS-CHAVE NESTE CAPÍTULO

- Coesão alta
- Classes de especificação
- Quantidade
- Medida
- Estratégia
- Composição
- Hierarquia organizacional
- Conta/transação
- Intervalo
- Padrões temporais

7.1 Introdução aos padrões de modelagem conceitual

Diagramas de classe que representam modelos conceituais quase sempre se tornam mais complexos e difíceis de manter do que os analistas e outros membros da equipe poderiam gostar. Existem técnicas que reduzem a complexidade desses diagramas e ao mesmo tempo melhoram sua expressividade. Modelos ingênuos podem ser desnecessariamente complexos.

Essas técnicas são chamadas de *padrões de análise* e podem ser entendidas como um caso especial de *padrões de design* (Gamma; Helm; Johnson; Vlissides, 1995) que são aplicados especificamente ao modelo conceitual. Vários padrões apresentados neste capítulo foram inicialmente descritos por Fowler (2003).

Padrões de análise ou design não são regras que devem ser obedecidas, mas recomendações baseadas em experiência prévia. É responsabilidade do analista decidir quando aplicar um dado padrão aos modelos.

7.2 Coesão alta¹

Coesão alta é tão importante para modelagem orientada a objetos que, muitas vezes, é considerado mais como um princípio ou axioma do que um padrão propriamente dito.

¹ Usualmente, coesão alta é mencionada em conjunto com *acoplamento baixo*. Entretanto, como problemas de acoplamento ocorrem quando colaborações entre objetos são projetadas, a discussão sobre acoplamento baixo é deixada para a Seção 9.6.

Um conceito com coesão alta é mais estável e reusável do que um conceito com coesão baixa, o qual pode se tornar rapidamente confuso e difícil de manter. A maioria dos sistemas poderia ter toda sua informação representada em uma única tabela, mas esse seria um caso extremo de coesão baixa e não seria nada prático.

Anteriormente, mencionamos que conceitos não devem ter atributos que pertencem a outros conceitos (por exemplo, um carro não deve ter o CPF seu dono como um de seus atributos). Atributos também não devem ser tipados com nomes de classes ou estruturas de dados (listas, conjuntos, *arrays* etc.) porque todas essas situações são evidências de baixa coesão (uma classe com este tipo de atributo provavelmente estaria representando mais do que um único conceito). Por exemplo, uma instância de *Pedido* não deveria ter um atributo como *listaDeItens* porque os itens têm seus próprios atributos e associações; eles deveriam aparecer relacionados ao *Pedido* por uma associação de 1 para *.

No exemplo da Figura 6.65, quando foi descoberto que um comprador teria mais do que um endereço, em vez da solução ingênua de criar atributos como *endereço1*, *endereço2* etc., ou mesmo *endereços:Array<String>*, foi criado um novo conceito (*Endereço*), o que foi associado ao comprador. Imagine como seria difícil lidar com informações como *taxaDeEntrega* se os endereços fossem modelados como um *array* de *strings*.

Conceitos e atributos devem ter uma estrutura simples com elementos de alta coesão. Evidências de baixa coesão incluem cenários nos quais alguns atributos de uma classe podem ser nulos dependendo dos valores de outros atributos. Se restrições complexas (invariantes) são necessárias para manter um conceito consistente, isso é equivalente a usar fita adesiva para manter as peças e um vaso quebrado juntos.

Adiante, na Figura 7.1, os atributos *valorPago* e *dataDePagamento* são mutuamente dependentes: ambos são nulos ou não nulos. Uma invariante deve ser adicionada à classe para evitar a situação na qual um deles é nulo enquanto o outro não é, para que se evitem instâncias inconsistentes. Invariantes são boas para representar regras de negócio que devem ser sempre válidas. Mas quando uma invariante apenas relaciona dois atributos de uma classe, isso é uma pista forte de que ela pode estar com falta de coesão.

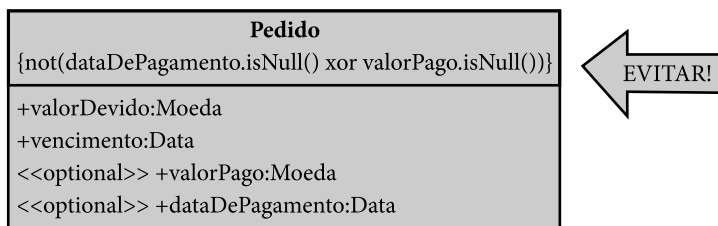
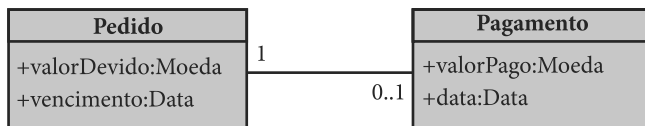
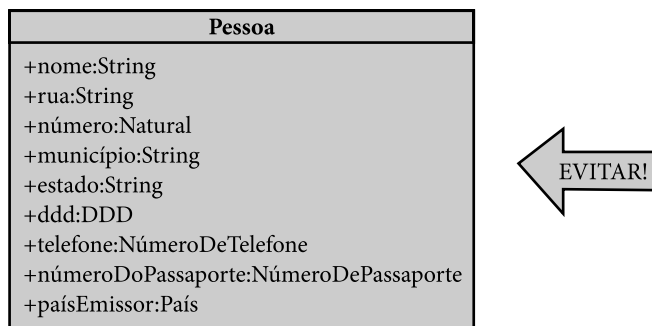


Figura 7.1

Uma classe com baixa coesão já que atributos dependem uns dos outros.


Figura 7.2

Uma solução de modelagem com classes de alta coesão.


Figura 7.3

Uma classe com baixa coesão em razão de atributos fortemente relacionados.

Uma forma melhor de modelar essa situação é mostrada na Figura 7.2, a seguir, na qual os conceitos *Pedido* e *Pagamento* aparecem separados, mas com alta coesão. Neste caso, não há atributos opcionais dependendo uns dos outros e *não há necessidade de restrições*.

Outro problema potencial relacionado à coesão baixa é a existência de grupos de atributos fortemente correlacionados, como mostrado na Figura 7.3, na qual se pode observar que grupos de atributos têm relações mais fortes dentro do grupo, tais como *rua*, *número*, *município* e *estado* que compõem um endereço, ou *DDD* e *telefone* que são parte de um número de telefone completo, ou ainda *númeroDoPassaporte* e *paísEmissor*, que são parte da informação sobre o passaporte. A relação que existe dentro de cada grupo aparece implicitamente na ordem em que os atributos são apresentados na classe. Isso é ruim, porque, se os atributos forem ordenados de forma alfabética, por exemplo, a fraca estrutura dos grupos pode ser mutilada: *DDD*, *estado*, *município*, *número*, *númeroDoPassaporte*, *paísEmissor*, *rua* e *telefone*. A compreensão de um conceito não deveria depender da forma como seus atributos são ordenados.

Uma solução para melhorar a coesão deste modelo é mostrada adiante na Figura 7.4. Ela também abre o caminho para outras possibilidades de modelagem, tais como, por exemplo, permitir que uma pessoa tenha mais de um endereço ou mais de um telefone. Nesses casos, deve-se simplesmente mudar a multiplicidade da associação.

Ainda outra situação ocorre quando alguns atributos repetem seus valores para um grupo de instâncias. A Figura 7.5 apresenta um exemplo: os atributos *nomeDoComprador*, *cpfDoComprador* e *dataDeNascimentoDoComprador* repetem os mesmos valores para diferentes pedidos quando o comprador é o mesmo.

Esse tipo de situação pode ser eliminado pela divisão do conceito em dois conceitos associados com coesão melhor, como adiante na Figura 7.6. Note que mesmo nomes de atributos são encurtados com esta escolha de modelo.

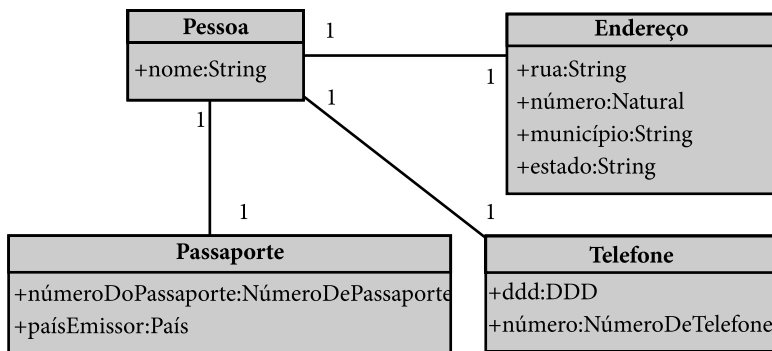


Figura 7.4

Um modelo para o exemplo da Figura 7.3 com coesão mais alta.

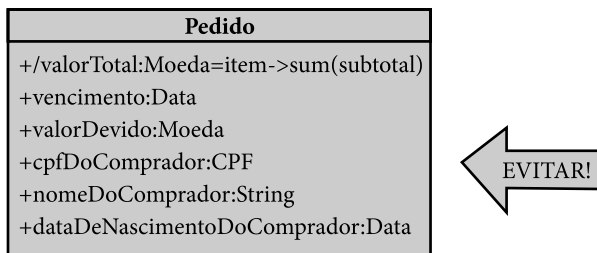


Figura 7.5:

Uma classe com baixa coesão em razão de atributos que aparecem repetidos em diferentes instâncias.

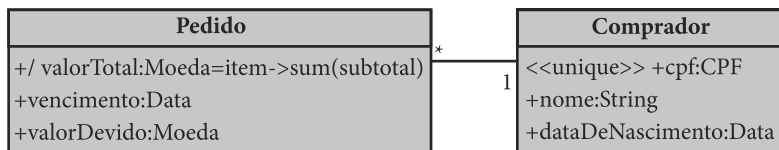


Figura 7.6

Um modelo para o exemplo da Figura 7.5 com classes de alta coesão.

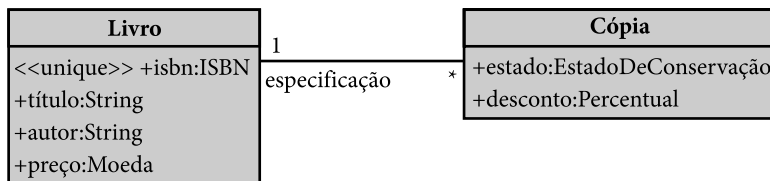


Figura 7.7

Uma classe e sua classe de especificação.

7.3 Classes de especificação

Um caso especial de coesão baixa ocorre quando um objeto é confundido com sua especificação. Esta situação é muito frequente: algumas vezes, produtos ou itens físicos compartilham uma especificação em comum, enquanto seus exemplares podem ter algumas diferenças entre eles.

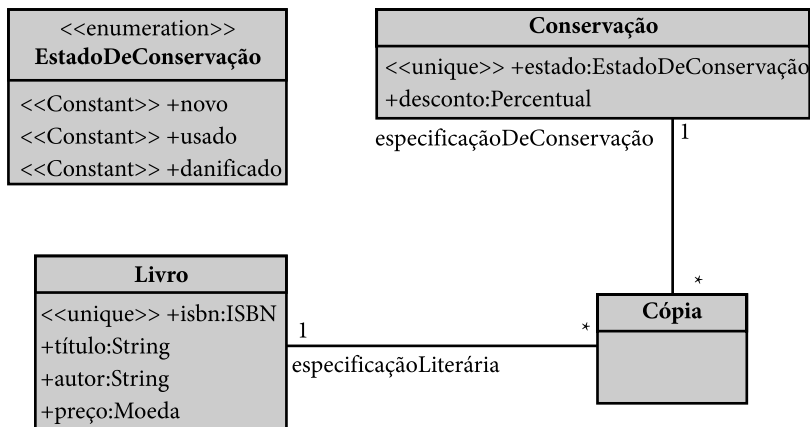
Um exemplo de especificação já discutido no exemplo Livro é a diferença entre o conceito de um livro enquanto *título publicado* e um livro como uma *cópia física* de um título publicado. Se estes dois conceitos de livro não forem distinguidos, muitos problemas poderão ocorrer. Nos exemplos anteriores, a classe *Livro* se referia ao título publicado, enquanto *Item* corresponde a um conjunto de cópias físicas. Assim, uma instância de *Livro* é uma *especificação* para um conjunto de instâncias não distinguíveis de *Item*.

Outra possibilidade seria considerar cada item como uma única cópia de um livro. Isso eliminaria a necessidade de incluir o atributo *quantidade* na classe *Item*, mas também criaria mais instâncias e associações para um pedido, caso ele incluísse mais do que uma cópia do mesmo livro.

Dependendo das necessidades de informação, seria necessário diferenciar uma cópia de outra. Por exemplo, se livros usados serão vendidos, o estado de conservação deveria ser mantido para cada cópia. Nesse caso, em vez de um conjunto de itens, a classe *Livro* deveria ser usada como especificação da classe *Cópia*, cujas instâncias são cópias individuais de um livro.

A classe de especificação *Livro* deveria conter os atributos e associações que não variam para diferentes cópias. Por exemplo, diferentes cópias do mesmo livro têm o mesmo título, autor, ISBN, preço base etc. Entretanto, o estado de conservação e desconto podem ser distintos para cada uma delas. Assim, esses atributos (e associações, se houverem) são colocados na classe *Cópia*, como mostrado na Figura 7.7.

É possível que uma classe tenha mais do que uma especificação. Por exemplo, o desconto do livro poderia ser predeterminado dependendo do estado de conservação da cópia (uma enumeração com valores como *novo*, *usado*, *danificado* etc). Assim, uma cópia, além de ser especificada pelo título publicado, também pode ser especificada pelo seu estado de conservação. Assim, *Conservação* seria uma classe com algumas instâncias que podem especificar cópias físicas de livros, e definir seus descontos como mostrado a seguir na Figura 7.8.

**Figura 7.8**

Uma classe com duas classes de especificação.

As instâncias da Classe *Conservação* têm dois atributos. O primeiro é o estado de conservação, que é um valor obtido da enumeração *EstadoDeConservação*. Existem inicialmente apenas três valores possíveis: *novo*, *usado* e *danificado*. O segundo atributo é o percentual de desconto que poderia ser aplicado a uma cópia dependendo de seu estado de conservação. Como é esperado que livros com o mesmo estado de conservação tenham o mesmo desconto, então, o atributo *estado* na classe *Conservação* deve ser único e, portanto, marcado com `<<unique>>`. Caso contrário, diferentes descontos poderiam ser associados ao mesmo estado de conservação.

7.4 Quantidade

Frequentemente, a equipe enfrenta a necessidade de modelar quantidades que não são meramente números. Por exemplo, o peso de um livro poderia ser definido como 400. Mas 400 o que? Gramas? Libras? Quilos? Uma solução é definir um tipo específico de peso (*Gramas*, por exemplo), e então usá-lo consistentemente. O atributo então poderia ser declarado como *peso:Gramas*. Mas isso exigiria que todos os livros tivessem seu peso expresso em gramas.

Em alguns casos, o sistema deve poder ser configurável para suportar diferentes unidades de peso. Em alguns países, gramas e quilogramas são usados, enquanto outros usam libras, ou mesmo outras unidades. Se o atributo tem um tipo que se refere a uma unidade específica, então o sistema deve sofrer refatoração para acomodar as novas unidades.

Entretanto, o padrão *Quantidade* permite que diferentes sistemas de unidades coexistam e sejam facilmente intercambiáveis. O padrão consiste em criar um novo tipo primitivo *Quantidade*, o que tem dois atributos, como mostrado na Figura 7.9.

Dessa forma, o peso de cada livro é especificado como uma quantidade primitiva formada por um *valor* numérico e uma *unidade*, cujo tipo é uma enumeração com valores possíveis definidos como *gramas*, *libras* e *quilos*.

Se uma conversão entre as unidades for necessária, uma opção é transformar a enumeração *Unidade* em uma classe normal e associar uma nova classe *Razão* a duas unidades, *origem* e *destino*, como mostrado a seguir na Figura 7.10. Quando uma quantidade da unidade original deve ser convertida em uma quantidade da unidade destino, então o valor da quantidade associado à origem deve ser dividido pela razão de conversão.

Assim, por exemplo, uma instância de *Unidade* cujo nome é “*gramas*” poderia ser ligada a uma instância de *Razão* como *origem*, a qual poderia ser ligada a outra instância de *Unidade* denominada “*quilos*”. O *valor* da razão deveria ser 1000, porque para converter gramas em quilos o valor original em gramas deve ser dividido por 1000.

7.5 Medida

Uma evolução do padrão *Quantidade* é o padrão *Medida*, o qual pode ser usado quando várias diferentes medidas sobre um objeto devem ser tomadas possivelmente em diferentes tempos. Por exemplo, uma pessoa em observação em um hospital pode ter várias medidas tomadas de tempos em tempos: temperatura do sangue, pressão sanguínea, nível de glicose no sangue etc. Milhares de diferentes medidas poderiam ser tomadas, mas usualmente apenas umas poucas são realmente tomadas para cada paciente. Portanto, para evitar criar um conceito com milhares de atributos sendo que a maioria deles seria provavelmente nulo, uma opção melhor é usar o padrão *Medida*, como mostrado na Figura 7.11.

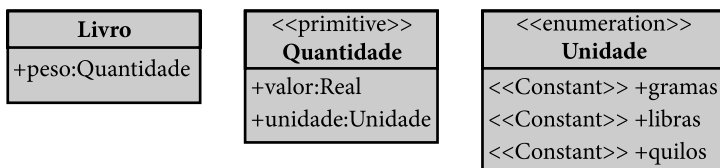


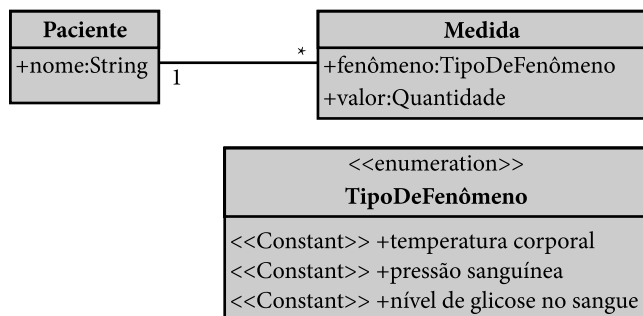
Figura 7.9

Definição e uso de *Quantidade*.



Figura 7.10

Unidade com razão de conversão.

**Figura 7.11**

Definição e uso do padrão *Medida*.

Assim, um paciente pode ter uma série de medidas tomadas, cada uma das quais avaliando um diferente fenômeno e apresentando um valor que corresponde a uma quantidade (seguindo o padrão *Quantidade*).

Ainda é possível tornar esses padrões um pouco mais sofisticados pela adição de atributos na classe *Medida* para indicar o instante de tempo no qual a medida foi tomada, e também a validade da medida. Por exemplo, o fato de que um paciente teve febre há algumas horas atrás não significa que ele ainda esteja com febre. Esta evolução é explicada na Seção 7.12.1.

Outra evolução para este padrão é adicionar um atributo para definir a *precisão* da medida tomada. Por exemplo, a temperatura corporal usualmente é medida com uma precisão de 0,1 graus (Celsius ou Fahrenheit).

7.6 Estratégia

Foi mencionado anteriormente que um dos grandes desafios dos requisitos é gerenciar sua mudança. Especialmente requisitos transitórios devem ser acomodados no design do sistema de forma que sua mudança provoque o mínimo impacto no sistema.

Alguns casos são relativamente simples de tratar. Por exemplo, se já foi decidido que o sistema deve operar em diferentes países, o padrão *Quantidade* poderia ser usado para lidar com moedas e outras medidas que podem variar de país para país.

Mas existem situações muito mais complicadas. Por exemplo, os procedimentos de cálculo de impostos podem variar muito de país para país e mesmo de Estado para Estado em alguns países. Há alguns impostos que são calculados em relação ao lucro, outros em relação ao preço de venda, e assim por diante. Sistemas devem estar preparados para lidar com isso, mas as mudanças são completamente imprevisíveis. Apenas configurar parâmetros usualmente não é suficiente.

Outro exemplo é a política de desconto da livraria. A política corrente poderia ser aplicar 10% de desconto em pedidos acima de R\$ 100,00, por exemplo. Entretanto, após algum tempo, políticas criativas e imprevisíveis poderiam ser definidas pelo departamento de vendas, tais como:

- Dar um livro grátis de até R\$ 50,00 para pedidos acima de R\$ 300,00.
- Dar 20% de desconto em até dois livros no dia do aniversário do comprador.
- Dar 5% de desconto em livros de terror na Sexta-Feira 13.
- O comprador gira uma roleta e ela define o desconto que será dado, de 1% a 10%.

Além disso, seria possível combinar políticas, se aplicável, ou escolher as que dão o maior desconto.

O padrão *estratégia* sugere que nesses casos o procedimento (por exemplo, calcular impostos ou descontos) deve ser separado dos dados aos quais ele se aplica. Em outras palavras, se um desconto é aplicado a um pedido, então o desconto não deve ser meramente um método implementado na classe *Pedido*. O desconto deve ser definido como algo mais fácil de mudar. A solução proposta pelo padrão *Estratégia* é criar uma classe abstrata associada ao pedido; essa classe abstrata pode ter subclasses concretas que representam políticas de desconto concretas, como mostrado adiante na Figura 7.12.

Assim, cada instância de *Pedido* é associada a uma instância de uma ou mais subclasses da estratégia *Desconto*.

UML não permite que um atributo seja declarado como *{abstract}*; apenas classes e métodos podem ser declarados assim. Entretanto, como atributos derivados são implementados como métodos no final, a ideia de um *atributo derivado abstrato* estereotipado com *<<abstract>>* como mostrado na Figura 7.12 é coerente com o espírito da UML.

Assim, a classe abstrata *Desconto* implementa um atributo derivado abstrato *valor*, o qual é implementado de forma concreta em cada uma de suas subclasses. Se alguma das estratégias necessitar de dados do comprador ou do pedido para calcular o desconto, esses dados podem ser acessados por meio das associações da classe abstrata de desconto para as classes que possuem a informação.

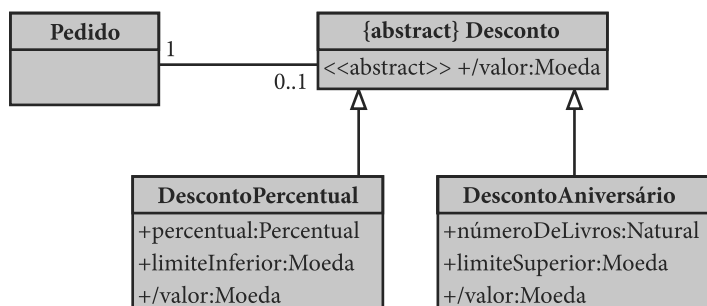
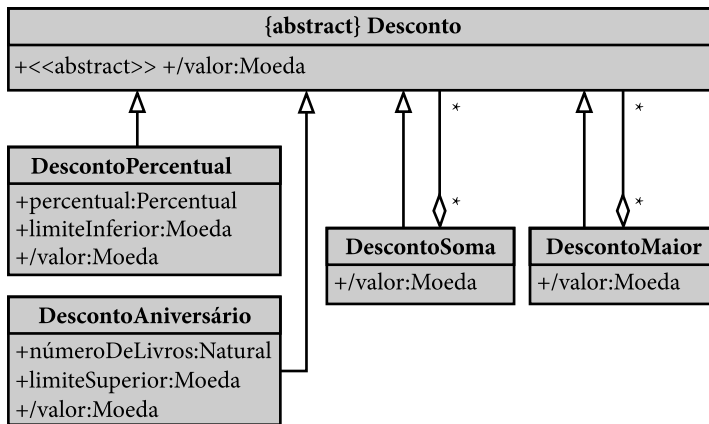


Figura 7.12

Padrão *Estratégia*.

**Figura 7.13**

Exemplo do padrão *Composição*.

Este padrão minimiza dois problemas relacionados à mudança de requisitos. Primeiro, ele mantém a estratégia de desconto aplicada quando um pedido é emitido, mesmo que as estratégias mudem no futuro. Segundo, se novas estratégias são criadas depois, é suficiente implementar uma ou mais subclasses de *Desconto*. Isso não afeta as antigas estratégias, nem os pedidos antigos.

7.7 Composição²

Como mencionado na Seção 7.6, algumas vezes, diferentes estratégias precisam ser agrupadas e uma combinação ou seleção delas deve ser aplicada. A combinação de diferentes estratégias de desconto pode ser obtida, por exemplo, pela soma dos valores obtidos para o atributo derivado *quantidade* para cada estratégia individual, ou pela escolha do maior ou menor dentre os valores. A escolha de uma ou outra combinação produz diferentes estratégias agregadas.

Na Figura 7.13, são mostradas duas estratégias de composição. *DescontoSoma* define *valor* como a soma dos valores retornados por cada componente do desconto:

```
Context DescontoSoma::valor:Moeda
derive:
    self.desconto->sum(umDesconto|umDesconto.valor)
```

DescontoMaior seleciona o maior desconto entre cada um dos elementos agregados:

```
Context DescontoMaior::valor:Moeda
derive:
    self.desconto->maiorElemento(umDesconto|umDesconto.valor)
```

² *Composição* é o nome clássico para este padrão. Entretanto, se os elementos básicos puderem ser compartilhados entre diferentes elementos da composição, trata-se de uma associação de *agregação* que está sendo usada em vez da *composição*. Isso acontece quando estratégias são agregadas.

A versão corrente de OCL não define a função *maiorElemento* que deveria tomar o maior elemento de uma coleção, de acordo com o critério colocado entre parênteses, mas ela poderia ser definida como uma nova operação. Outra solução seria definir a expressão usando funções que já são fornecidas pela linguagem, tal como:

```
Context DescontoMaior::valor:Moeda
derive:
    self.desconto->sortedBy(umDesconto|umDesconto.valor)->last()
```

O leitor pode achar que não é necessário produzir uma versão ordenada de uma lista inteira apenas para obter seu maior elemento. Mas lembre-se de que isso é uma especificação, não implementação; é o resultado final que conta. De qualquer forma, definir uma nova operação nesse caso, se possível, seria melhor porque o significado da operação seria mais claro.

7.8 Hierarquia organizacional

Outra situação comum consiste na necessidade de representar hierarquias organizacionais. É comum, por exemplo, representar a organização administrativa de uma empresa como uma hierarquia organizacional como a mostra a Figura 7.14:

Entretanto, hierarquias como esta usualmente não se comportam bem. Primeiro de tudo, este tipo de organização não é seguido por qualquer empresa: outras empresas podem ter diferentes níveis ou diferentes composições entre eles. Em segundo lugar, a estrutura da empresa pode mudar ao longo do tempo. Novas divisões ou escritórios podem ser criados, alguns podem ser unificados e outros divididos. Finalmente, mas não menos importante, diferentes visões de uma organização podem existir ao mesmo tempo; por exemplo, o departamento de finanças pode ter uma visão diferente sobre a mesma organização.

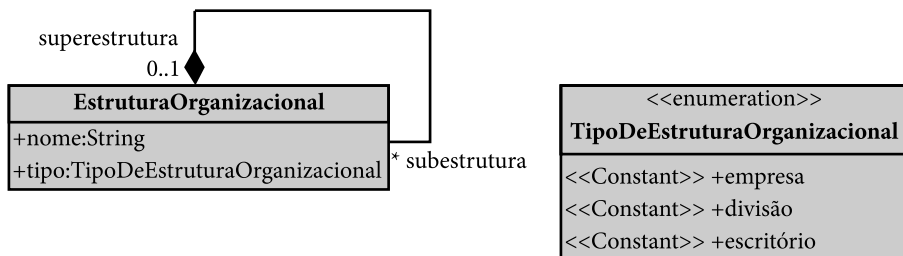
Como lidar com toda essa complexidade em um modelo conceitual? Pode-se usar o padrão *hierarquia organizacional*.

A solução consiste em não considerar os diferentes níveis de uma organização como conceitos, mas como instâncias de um único conceito, como mostra a Figura 7.15:



Figura 7.14

Representação direta de uma estrutura organizacional de uma empresa usando classes para os diferentes níveis de composição.

**Figura 7.15**

Aplicação do padrão hierarquia organizacional.

Dessa forma, ganha-se flexibilidade em relação a lidar simultaneamente com estruturas organizacionais de diferentes empresas. Além disso, eventuais mudanças na estrutura da organização, tais como adicionar mais níveis ou mudar dependências, podem ser mais fáceis de realizar.

Este padrão tem algumas variações que podem ser usadas se hierarquias concorrentes, estruturas que se sucedem no tempo, ou estruturas equivalentes forem também representadas. Todas essas variações são discutidas nas seções seguintes.

7.9 Junção de objetos

Uma das hipóteses das equipes de análise que muitas vezes falham é que os usuários vão fazer tudo certo. Isso nem sempre é o caso. Falhas do usuário (e algumas vezes sabotagem) são ainda frequentes, apesar dos esforços para construir interfaces a prova de falhas. Um usuário poderia, por exemplo, registrar uma nova editora no sistema, e mais tarde descobrir que ela já estava registrada. A introdução de um código incorreto ou a impossibilidade de ter uma identificação única para um objeto do mundo real pode causar esse tipo de situação. O resultado é que dois objetos são registrados no sistema, ambos representando a mesma editora, e cada uma delas possivelmente têm seu próprio conjunto de livros associados, alguns repetidos, outros não.

A solução quando esse tipo de erro acontece é juntar os objetos, usualmente copiando um sobre o outro. Essa operação pode realizar-se diretamente sobre o banco de dados, mas em algumas situações ela pode ser tão comum que requeira que o sistema esteja preparado para permitir essa possibilidade como uma funcionalidade de usuário.

Além disso, nem sempre é um *erro* que causa a necessidade de uma junção. Em algumas situações, há objetos considerados equivalentes por um grupo de pessoas e não equivalentes por outro grupo. Em outros casos, como nas hierarquias organizacionais com múltiplas visões, pode haver a necessidade de indicar que dois elementos da estrutura em diferentes visões são equivalentes, ou que um sucede ao outro no tempo.

As subseções seguintes apresentam as principais estratégias para lidar com esses tipos de situação.

7.9.1 Copiar e substituir

A primeira estratégia que vem à mente quando é necessário juntar dois objetos consiste em copiar os dados de um objeto sobre o outro (*copiar e substituir*). O comando copiar/substituir poderia ser definido por contrato (veja Capítulo 8), e a equipe deveria definir para cada atributo e cada associação o que acontece durante o comando de cópia/substituição. Regras deveriam ser definidas para decidir se um atributo vai ser copiado sobre outro, se seus valores serão adicionados ou se o valor mais novo ou maior deve prevalecer etc. Em relação às associações, a equipe deve decidir o que acontece: se uma substitui a outra ou se suas propriedades são adicionadas, e assim por diante.

O registro da data da última inclusão ou alteração de um conceito pode ser útil para decidir sobre qual atributo deve ser mantido no caso de um conflito. Por exemplo, se um comprador registrado faz um novo registro e isso tem de ser resolvido, o endereço mais recente deve ser mantido e não o mais antigo. No entanto, todos os pedidos devem ser adicionados à instância resultante.

Após a execução do comando copiar/substituir, a instância que foi copiada deve ser destruída e todas as referências para ela devem ser redirecionadas para a instância que recebeu os dados.

7.9.2 Sucessor

Sucessor é uma técnica que pode ser usada quando o objeto original for mantido e não destruído. Sucessor é aplicável, por exemplo, no caso de estruturas organizacionais que se sucedem no tempo. Suponha que os departamentos de marketing e vendas sejam reunidos em um único departamento de *contato com clientes*; os departamentos originais devem ser marcados como não mais ativos, e um novo departamento deve ser adicionado como seu sucessor. A estratégia de sucessão pode ser implementada por uma associação reflexiva como mostra a Figura 7.16:

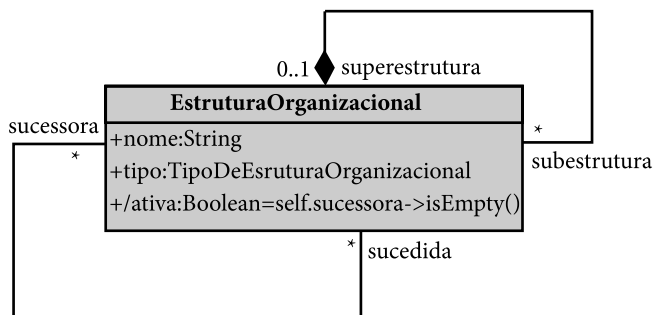
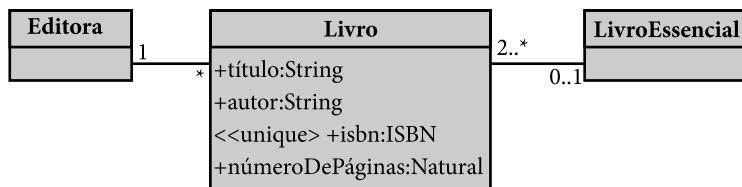


Figura 7.16

Um exemplo de estratégia de sucessão.

**Figura 7.17:**

Técnica Essência/Aparência.

O atributo derivado *ativa* indica se a estrutura é ativa ou se ela já foi sucedida. Ele é verdadeiro se o conjunto de sucessoras for vazio e falso caso contrário.

Manter a estrutura organizacional original mesmo se ela não for mais ativa pode ser importante para efeito de registro. Algum dia, alguém poderia querer saber quanto um dado departamento que não existe mais gastou em palitos de dente.

Poderia ser útil adicionar uma classe de associação à associação *sucessora/sucedida*; seus atributos poderiam indicar, por exemplo, a data na qual a sucessão ocorreu, ou o contexto ou visão na qual ela deve ser considerada. Se este tipo de informação for necessário, a equipe deveria considerar usar um dos padrões temporais (Seção 7.12).

7.9.3 Essência/aparência

Outra situação que pode ainda acontecer é a existência de objetos que são considerados equivalentes, mas devem ser mantidos como objetos separados. Isso não é um erro de registro e não é um objeto que sucede a outro: isso é *equivalência entre objetos*.

A equivalência entre objetos pode ser modelada pelo uso de um objeto essência que é associado a um conjunto de objetos equivalentes. Ao contrário de *copiar/substituir*, os objetos originais são mantidos, e ao contrário de *sucessor* não há objeto ativo ou sucedido: todos os objetos associados são equivalentes. A Figura 7.17 mostra um exemplo de uma classe (*Livro*) que aceita que seus membros compartilhem uma essência comum.

No exemplo, consideramos que o mesmo livro, em essência, especialmente um clássico, pode ser publicado por diferentes editoras. Cada publicação é distinta, com ISBN e número de páginas diferentes. O título e o nome do autor podem variar (nomes próprios podem ser abreviados ou traduzidos algumas vezes), mas a essência do texto é a mesma. Alguns usuários poderão estar interessados em adquirir *A República* de Platão de uma determinada editora, e outros poderão estar interessados no livro independentemente da edição ou editora. Assim, os usuários interessados em uma publicação específica estão buscando a aparência (uma determinada instância de *Livro*) e os usuários interessados no texto estão buscando a essência – a editora pouco importa. O segundo grupo de usuários, quando estiver visualizando qualquer edição de *A República*, deve também poder ver as outras instâncias de *Livro* que estiverem ligadas ao mesmo *LivroEssencial*, se ele existir. Em outras palavras, se você procurar pelo livro *umLivro* e se *umLivro.livroEssencial*-

->*notEmpty()*, então você verá *umLivro.livroEssencial.livro*, o que corresponde ao conjunto de livros que compartilha a mesma essência.

Nem *tudo Livro* deve estar ligado a um *LivroEssencial*, apenas aqueles que participam de uma classe de equivalência. Além disso, deve haver pelo menos dois livros equivalentes para criar um *LivroEssencial*. É por isso que o papel de associação de *LivroEssencial* para *Livro* é de 2..*.

Objetos são considerados equivalentes se eles são ligados pelo mesmo objeto essência. O objeto essência existe apenas para estabelecer a equivalência; usualmente ele não deve ter nenhuma outra propriedade. Caso contrário, talvez o padrão *Classe de especificação* devesse ser usado em seu lugar.

7.9.4 Desfazendo uma junção

Justamente quando você pensa que o mundo real não pode ficar mais complexo, ele fica. Portanto, se houver a possibilidade de juntar objetos, pode também ser possível ter que separar os objetos novamente. Esses comandos podem ser realizados diretamente no banco de dados ou por meio de operações bem planejadas em nível de interface com usuário de sistema.

Para permitir que as junções que usam a técnica *copiar/substituir* sejam desfeitas, um *backup* dos objetos originais deve ser mantido porque a técnica destrói um dos objetos e desfigura o outro. A técnica *sucessor* permite desfazer a junção simplesmente pela remoção da associação. No caso de *essência/aparência*, é necessário remover a associação (e remover também o objeto essência se restarem menos do que duas ligações).

Entretanto, em todo o caso, é importante decidir como lidar com eventuais mudanças em um objeto que ocorreram enquanto ele estava ligado com outros.

Uma forma de implementar a possibilidade de desfazer mudanças e quaisquer outras operações é usar uma variação dos padrões temporais (Seção 7.12.3) que mantém o registro de antigos valores de atributos e associações de objetos.

7.10 Conta/transação

O padrão *conta/transação* é fortemente relacionado à área de negócios, mas tem aplicação variada. Foi mencionado anteriormente que livros podem ser catalogados, pedidos, recebidos, enviados, descartados etc. Tais movimentações, bem como as transações financeiras envolvidas, dão origem a conceitos tais como *Pedido*, *Envio*, *Descarte*, *Devolução* etc., cada um com seus próprios atributos e associações.

Entretanto, é possível identificar um núcleo comum para todos esses conceitos e muitos mais, o qual é constituído por um simples e poderoso padrão.

Uma *Conta* é um conceito que possui quantidades de *alguma coisa* (como itens, produtos ou dinheiro). Uma *Conta* tem um saldo que usualmente consiste da soma de todos os depósitos e retiradas.

No entanto, depósitos e retiradas são usualmente apenas movimentações de bens ou dinheiro de uma conta para outra. Assim, uma *Transação* consiste de duas movimentações: um depósito em uma conta e uma retirada do mesmo valor de outra conta. A Figura 7.18 ilustra essas classes.

Uma transação consistente com duas movimentações como na Figura 7.18 precisa de duas movimentações com o mesmo valor absoluto, mas sinais opostos. Em outras palavras, se uma transação retira R\$ 5,00 de uma conta, ela deve necessariamente depositar R\$ 5,00 em outra conta; a retirada tem sinal negativo e o depósito sinal positivo. Assim, a classe *Transação* necessita da seguinte invariante:

```
Context Transação
  inv:
    movimentação->sum(valor)=0
```

Isso significa que para cada instância de *Transação* a soma do valor das duas movimentações associadas deve ser zero.

O atributo derivado *saldo* da classe *Conta* é definido como a soma de todas as movimentações ligadas à *Conta* específica:

```
Context Conta:saldo
  derive:
    movimentação->sum(valor)
```

Várias situações relacionadas ao exemplo da livraria poderiam ser modeladas a partir de um conjunto de instâncias da classe *Conta*, tais como:

- Para cada *editora*, há uma instância de *Conta* da qual livros são “retirados”, isto é, ela é uma *conta de entrada* e seu saldo se torna cada vez mais negativo à medida que livros são pedidos para esse fornecedor.
- Há uma conta para *pedidos esperados*, que contém os livros que foram pedidos do fornecedor mas que ainda não chegaram.
- Há uma conta para *estoque*, que contém os livros disponíveis.
- Há uma conta para *livros vendidos*, que contém livros vendidos, mas não ainda enviados.
- Há uma conta para *livros enviados*, que contém livros enviados, mas cuja entrega ainda não foi confirmada pelo comprador.
- Há uma conta para *entregas confirmadas*, que contém livros enviados e confirmados pelo comprador. Seu saldo representa o conjunto total de livros que o comprador já comprou da livraria.

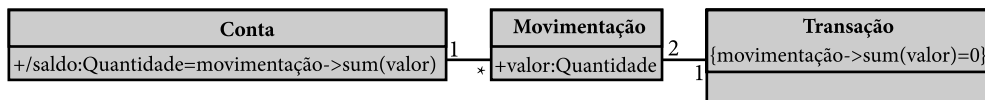
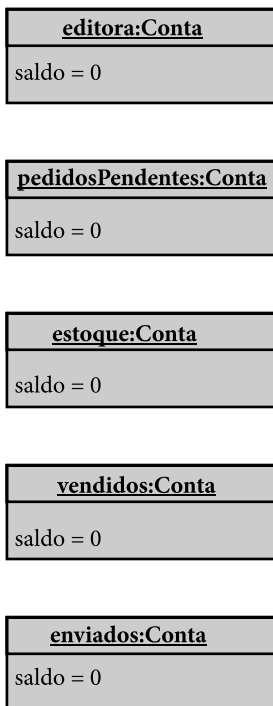


Figura 7.18

Classes para o padrão Conta/transação.

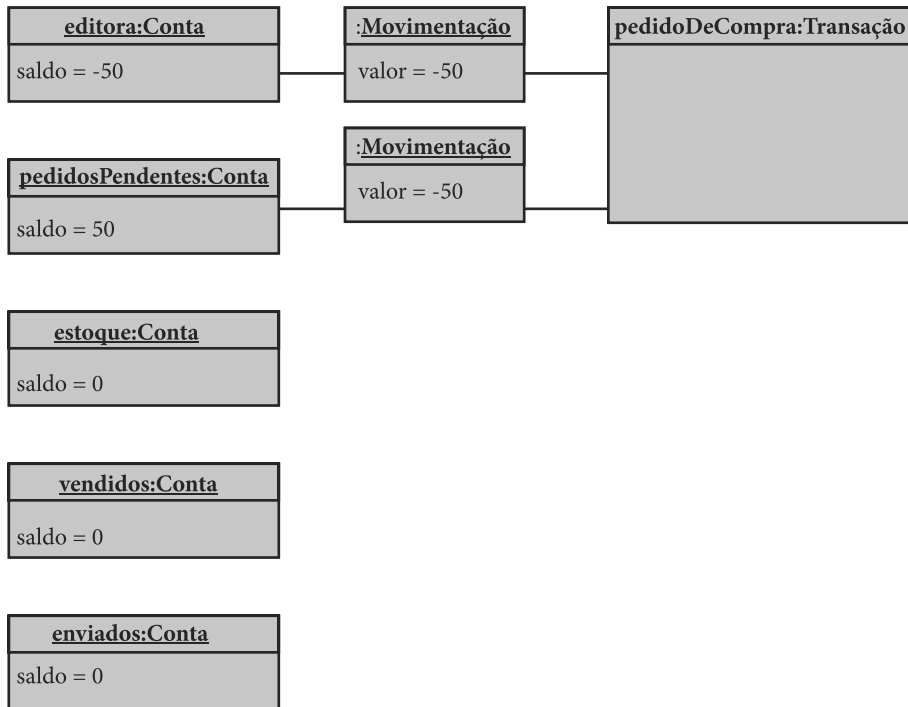

Figura 7.19

Estado inicial de um sistema de contas para produtos.

Paralelamente às transações com livros, há transações concomitantes com dinheiro. Existem contas a receber, contas a pagar, pagamentos recebidos, contas pagas, investimentos, débitos, valores reservados para pagar impostos etc.

Assim, as muitas transações do exemplo da livraria poderiam ser modeladas como instâncias de *Transação*. Por exemplo:

- Um *pedido* é uma transação que retira livros de uma conta de *editora* e deposita em uma conta de *pedidos pendentes*.
- A *chegada* dos livros é uma transação que retira da conta de *pedidos pendentes* e adiciona a uma conta de *estoque*.
- Uma *venda* é uma transação que retira da conta de *estoque* e adiciona na conta de *livros vendidos*.
- Um *envio* é uma transação que retira da conta de livros vendidos e adiciona na conta de *livros enviados*.
- Uma *devolução* é uma transação que retira de uma conta de *livros enviados* e adiciona na conta de *estoque* (se os livros não estiverem danificados).
- Uma *confirmação de entrega* é uma transação que retira de uma conta de *envio* e adiciona a uma conta de saída de *entregas confirmadas*.

**Figura 7.20**

Estado das contas após um pedido e compra de 50 livros ser emitido.

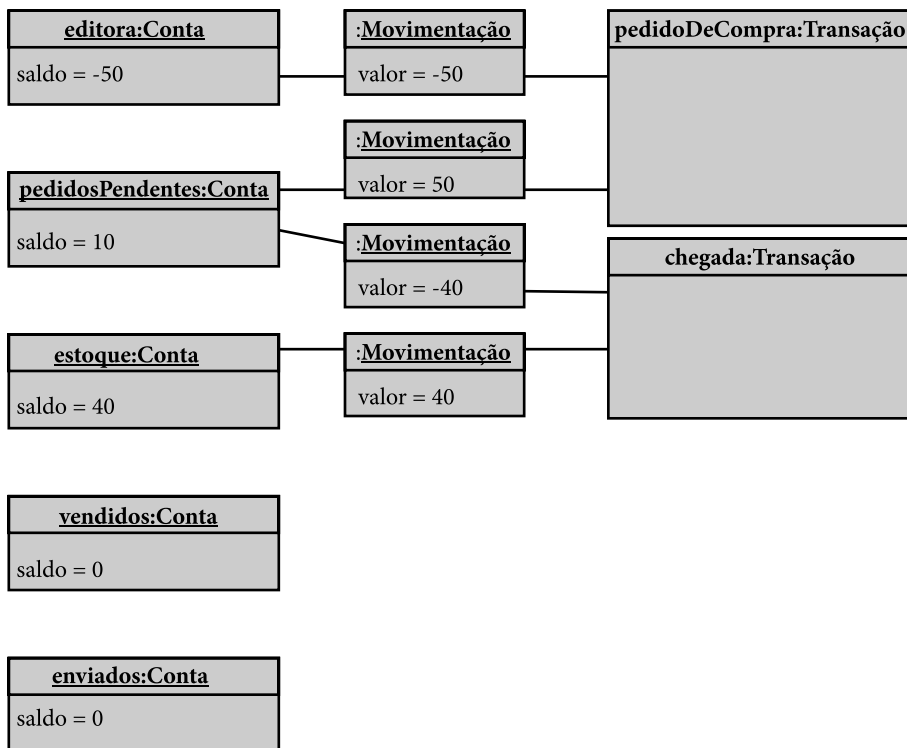
Como um exemplo, imagine uma empresa que foi aberta recentemente, na qual cinco instâncias³ da classe *Conta* foram criadas: *fornecedor*, *pedidosPendentes*, *estoque*, *vendidos* e *enviados*. A Figura 7.19 mostra o estado inicial desses objetos, no qual todas as contas têm saldo zero.

A Figura 7.20 mostra o conjunto de objetos resultante quando um *pedidoDeCompra* (instância de *Transação*) é criado. O pedido foi emitido para 50 livros.

A Figura 7.21 mostra o estado das contas se apenas 40 dos 50 livros pedidos chegarem.

A Figura 7.22 mostra o estado das contas após 25 livros serem vendidos.

³ Este exemplo lida com apenas cinco instâncias por simplificação. Muitas outras poderiam ser incluídas.


Figura 7.21

Estado das contas após 40 livros chegarem à livraria.

Novas transações e contas podem ser criadas dinamicamente (isto é, sem ter de recompilar o sistema). Por exemplo, uma nova transação pode ser criada para devolução de livros, a qual moveria os livros da conta *enviados* de volta para a conta de *estoque*. Além disso, uma nova transação para descartar livros poderia ser criada para mover livros da conta *estoque* ou *vendidos* para uma nova conta de livros *descartados*. Transações financeiras paralelas poderiam ser criadas da mesma forma pelo uso de *Conta* e *Transação*.

Este padrão é muito interessante como um exemplo de como uma ideia simples e poderosa pode lidar com tantas situações diferentes. Ele tem inúmeras variações e sofisticações possíveis. Por exemplo, *transações com várias pernas* podem ser criadas, se necessário, movendo de mais de uma conta para uma única conta; movendo de uma única conta para uma ou mais contas; ou movendo de mais de uma conta para mais de uma conta ao mesmo tempo. Para obter essa variação, a multiplicidade de papel de *Transação* para *Movimentação* na Figura 7.18 deveria ser modificada para 2..*.

E, finalmente, a Figura 7.23 mostra como as contas ficam após os 25 livros vendidos serem enviados.

Outra variação importante são as *movimentações mnemônicas*. Por exemplo, quando o dinheiro é recebido, impostos devem ser pagos, mas não necessariamente de forma imediata. Assim, quando uma transação movimenta dinheiro da conta de um cliente para a conta corrente da livraria, uma quantidade de dinheiro equivalente aos impostos devidos deve ser registrada em uma conta mnemônica de impostos. Quando for o momento de pagar os impostos, o saldo da conta mnemônica registra o total a ser pago.

Contas mnemônicas e contas normais poderiam ser classes distintas. Entretanto, como compartilham a mesma definição, elas podem ser mantidas como instâncias de uma única classe *Conta* na Figura 7.24. No entanto, movimentações mnemônicas devem ser distinguidas. Uma transação pode ter no máximo duas movimentações normais, mas um número indeterminado de movimentações mnemônicas (zero em muitos casos). Além disso, apenas movimentações normais estão sujeitas à invariante da classe *Transação*. Movimentações mnemônicas são ignoradas porque elas não movem realmente bens ou dinheiro.

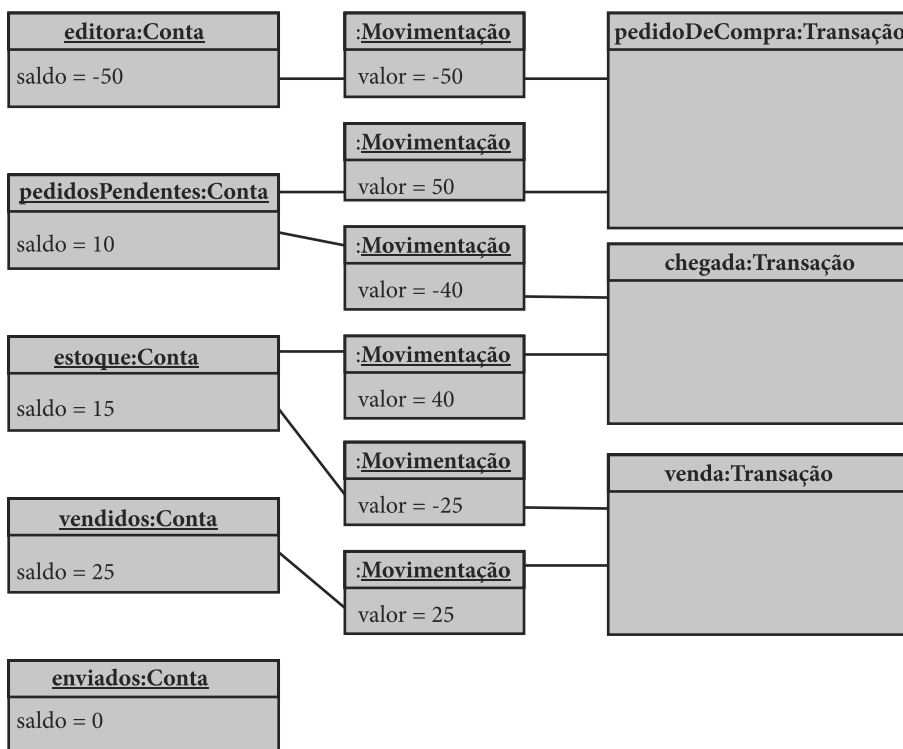


Figura 7.22

Estado das contas após 25 livros serem vendidos.

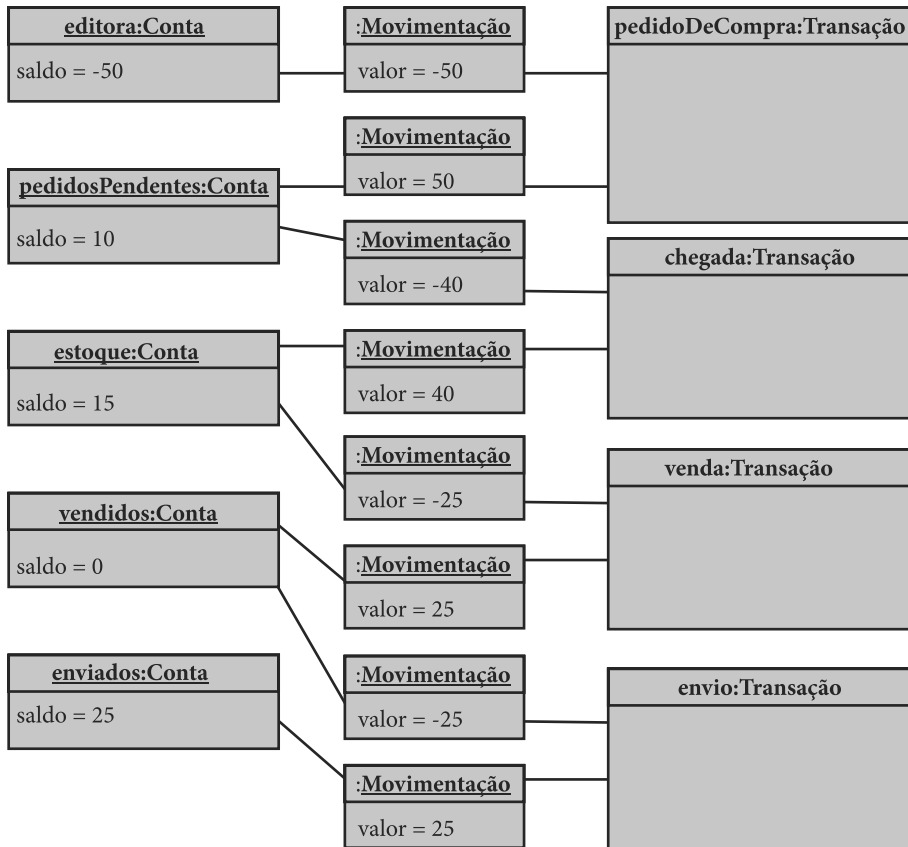


Figura 7.23

Estado das contas após 25 livros serem enviados.

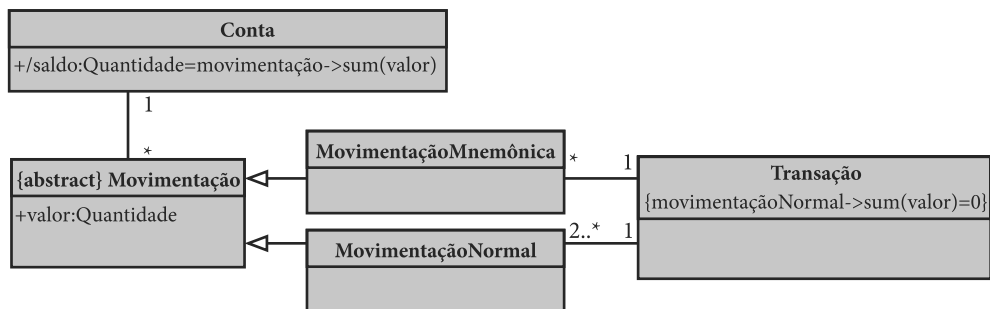


Figura 7.24

Evolução de *Conta/Transação* com movimentações mnemônicas.

Apesar de as contas normais e mnemônicas terem a mesma estrutura e comportamento, pode ser interessante diferenciá-las para evitar que o sistema possa fazer transferências normais para contas mnemônicas e vice-versa.

7.11 Intervalo

Quando um objeto tem um par de atributos indicando o início e o fim de algum fenômeno, como, por exemplo, uma data inicial e final, em vez de representar isso como dois atributos separados, seria melhor declarar um único atributo tipado como *Intervalo*, como mostra a Figura 7.25:

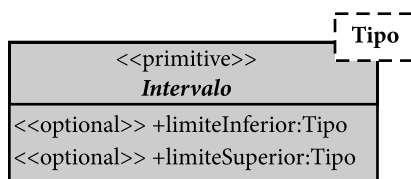


Figura 7.25

Um tipo primitivo genérico *Intervalo*.

O tipo primitivo *Intervalo* é parametrizado, o que significa que quando uma instância é criada ela deve indicar o *Tipo* a ser usado para definir seus atributos. Por exemplo, um intervalo sobre *Data* poderia ser criado como *Intervalo*[*Data*], e um intervalo sobre uma quantidade definida como *Distância* poderia ser definido como *Intervalo*[*Distância*].

Existem duas razões para usar um intervalo em vez de dois atributos:

- Se os limites inferior e superior forem atributos separados em uma classe conceitual, eles estarão fortemente relacionados, e isso vai contra o princípio da coesão alta explicado na Seção 7.2.
- Operações que são específicas a intervalos (por exemplo, verificar se um valor está dentro do intervalo) serão necessárias. Se houver um tipo primitivo *Intervalo* específico, essas operações podem ser implementadas uma única vez no tipo primitivo. Se dois atributos forem usados em vez disso, cada classe na qual eles forem declarados deverá implementar sua própria versão das operações sobre intervalos.

É muito mais razoável implementar operações apenas uma única vez em uma classe com alta coesão do que implementá-las várias vezes em classes com baixa coesão.

Como visto na Figura 7.25, ambos os atributos na classe *Intervalo* são opcionais, o que significa que intervalos podem ser indefinidos em qualquer de seus limites. Por exemplo, um intervalo que inicia em primeiro de janeiro de 2013 e não tem limite superior poderia ser definido como *limiteInferior*="01/01/2013" e *limiteSuperior*=null.

Evoluções desse padrão poderiam tomar em conta propriedades matemáticas de intervalos. Por exemplo, existem intervalos *abertos* e *fechados*: [0..10] é um intervalo que

inclui 0 e 10, enquanto (0..10) é um intervalo que exclui 0 e 10. Além disso, (0..10] exclui 0 mas inclui 10 e [0..10) inclui 0 e exclui 10. Esta evolução poderia ser implementada simplesmente pela adição de dois atributos booleanos à classe: *limiteInferiorAberto* e *limiteSuperiorAberto*.

7.12 Padrões temporais

Frequentemente, o analista se defronta com a necessidade de lidar com tempo. Por exemplo, pode ser necessário representar as relações entre pessoas e seus empregos; mas se a associação entre pessoas e empregos representa apenas o tempo presente, então a informação sobre empregos antigos será perdida quando alguém se demitir. Esta seção discute alguns padrões que lidam com essa importante noção⁴.

7.12.1 Efetividade

Quando um objeto é válido por algum tempo (por exemplo, uma medida de temperatura que possui um tempo no qual ela foi feita e um tempo no qual ela é considerada válida), então o padrão *Efetividade* pode ser usado. Ele consiste em declarar um atributo *efetivo* na classe, o qual é tipado com um intervalo, como mostra a Figura 7.26:

7.12.2 Histórico

Se a informação histórica deve ser armazenada sobre os estados passados de uma associação, então o padrão *Associação histórica* pode ser usado. Este tipo de associação pode ser identificado pelo estereótipo <<history>> como visto na Figura 7.27, a seguir.

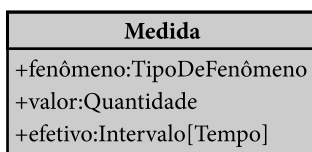


Figura 7.26

Uso do padrão *Efetividade*.

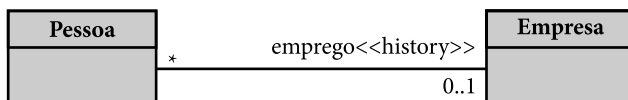
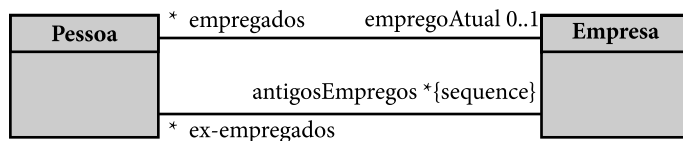


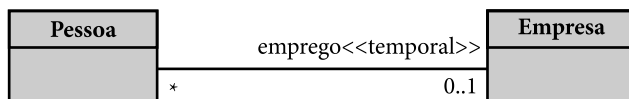
Figura 7.27

Um exemplo do padrão *Associação histórica*.

⁴ Martin Fowler apresenta um excelente resumo de vários padrões temporais, disponível em: <martinfowler.com/eaDev/timeNarrative.html>.

**Figura 7.28**

Modelo concreto para o estereótipo `<<history>>`.

**Figura 7.29**

Uma associação temporal.

O estereótipo da associação da Figura 7.27 significa que uma pessoa pode ter zero ou um emprego atualmente, mas ela pode ter tido vários empregos no passado. Esses empregos passados podem ser recuperados como itens em uma lista. Em outras palavras, é possível recuperar o último emprego, o penúltimo, e assim por diante.

Quando se chega ao design, cada classe com uma associação com papel navegável terá de implementar um método para acessar os elementos ligados. No exemplo da Figura 7.27, a classe de design para *Pessoa* teria um método *getEmprego()* que retorna a empresa na qual a pessoa correntemente trabalha ou o conjunto vazio se a pessoa estiver desempregada. Se a associação for estereotipada com `<<history>>`, em adição ao método padrão, haverá outro método com um parâmetro, *getEmprego(indice:Natural)*, no qual *getEmprego(1)* retornaria o emprego atual; *getEmprego(2)* retornaria o penúltimo emprego; *getEmprego(3)* retornaria o antepenúltimo, e assim por diante. Se não houver mais empregos para o índice dado, então o método retorna o conjunto vazio.

Na prática, este padrão pode ser modelado com duas associações, como mostra a Figura 7.28. O estereótipo então é uma forma de abreviar essa estrutura complexa, substituindo-a por uma mais simples.

7.12.3 Temporal

O padrão *Histórico* não é capaz de responder qual emprego a pessoa tinha em uma determinada data, ou se a pessoa esteve temporariamente desempregada no passado. Para representar esse tipo de informação, uma evolução deste padrão pode ser usada: uma associação que em adição à memória sequencial também tem memória temporal, como mostrado na Figura 7.29.

Uma associação temporal tem, em adição aos já mencionados métodos *get*, um método para obter o valor da associação em um dado momento do tempo. No exemplo, este

método poderia ser *getEmprego(data:Data)*. Assim, a classe *Pessoa* apresentada na Figura 7.29 teria pelo menos três métodos *getter* para o papel temporal *emprego*:

- *getEmprego()*: retorna o emprego corrente de uma pessoa ou o conjunto vazio se a pessoa estiver atualmente desempregada.
- *getEmprego(índice:Natural)*: retorna o emprego corrente ou um emprego antigo de uma pessoa dependendo do índice passado como argumento, ou o conjunto vazio se não houver emprego referente ao índice passado.
- *getEmprego(data:Data)*: retorna o emprego que uma pessoa tinha em uma determinada data ou o conjunto vazio se a pessoa estivesse desempregada naquela data.

Se for necessário saber o emprego de uma pessoa em uma determinada hora do dia, então em vez da *Data* o tipo primitivo *Tempo* deveria ser usado, o qual refere dia, mês, ano, hora, minuto e segundo. Para algumas aplicações, milissegundos, microssegundos, e assim por diante, poderiam também ser usados.

A Figura 7.30 apresenta o modelo para implementar o estereótipo *<<temporal>>*.

A classe *Emprego* não tem um par de atributos como *dataDeContratação* e *dataDeDemissão*, mas um único atributo *período* com o tipo *Intervalo[Data]*, representando um intervalo entre duas datas.

Não apenas associações podem ser temporais, mas atributos também. Se um atributo deve manter registro de seus valores anteriores, então o estereótipo *<<temporal>>* pode ser usado para manter o registro deles.

Por exemplo, alguns países permitem que as pessoas troquem seus nomes após casar, divorciar ou por decisão judicial. Na Figura 7.31, o CPF e a data de nascimento de uma pessoa são marcados como atributos que não podem mudar, ou, se mudarem, os valores antigos não são mantidos. Entretanto, nomes antigos devem ser preservados.

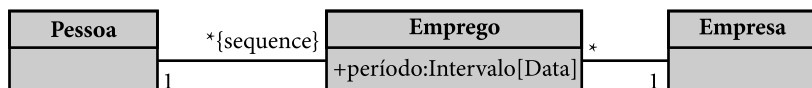


Figura 7.30

Uma possível implementação para o estereótipo *<<temporal>>*.

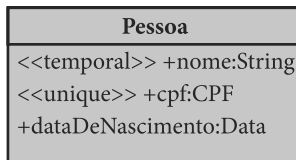


Figura 7.31

Padrão temporal aplicado a um atributo.

Outra variação deste padrão consiste em definir uma classe temporal. No caso de uma classe estereotipada com `<<temporal>>`, todos os seus atributos e papéis de associação são temporais. A classe também poderia fornecer um método para produzir uma versão de suas instâncias em um determinado instante de tempo; por exemplo, se *x* é uma instância de uma classe temporal, alguém poderia obter um estado prévio de *x* usando uma função como *x.noInstante(umInstanteDeTempo)*. Cada vez que um dos valores dos atributos ou ligações for alterado, uma nova versão do objeto é criada.

7.12.4 Bitemporal

Não apenas na ficção científica, mas também em muitos sistemas de informação atuais, o tempo pode ser considerado bidimensional. Existe uma dimensão do tempo na qual os eventos ocorrem e outra dimensão na qual nosso conhecimento sobre os eventos ocorre.

Volte a olhar para a Figura 7.29 e imagine que o sistema registra que Maria está atualmente trabalhando para a Empresa X. Agora imagine que no dia 11 de agosto descobrimos que de fato Maria mudou de emprego em 1º de março. Entretanto, nós apenas soubemos da mudança em 11 de agosto. Quaisquer ações que a empresa tenha feito entre 1º de março e 11 de agosto foram feitas com a informação de que Maria ainda trabalhava para a Empresa X. Algumas vezes, especialmente para efeito de contabilidade ou legislação, saber quando um registro foi feito é muito importante. O padrão bitemporal é uma forma elegante de manter o registro não só da linha de eventos, mas também da linha de nosso conhecimento sobre esses eventos (Figura 7.32).

Agora um novo *getter* pode ser introduzido para o papel *emprego*: *getEmprego(data, dataDoConhecimento:Data)*. O primeiro argumento é o instante de tempo no qual estamos interessados e o segundo argumento é o instante de tempo de nosso conhecimento sobre isso. Por exemplo, se quisermos responder a questão “em 15 de junho, onde acreditávamos que Maria trabalhasse no dia 1º de junho?” e se o ano for 2013, poderíamos usar *getEmprego*(“01/06/2013”, “15/06/2013”) para obter a resposta, a qual deveria ser “Empresa X”.

Se quiséssemos saber onde Maria trabalhava em 1º de junho baseados em nosso conhecimento corrente, poderíamos usar o *getter* mencionado no padrão temporal (Seção 7.12.3): *getEmprego*(“01/06/2013”), que é uma forma abreviada para *getEmprego*(“01/06/2013”, *hoje*).

A Figura 7.33 apresenta um possível modelo para implementar este estereótipo.



Figura 7.32

Padrão Bitemporal.

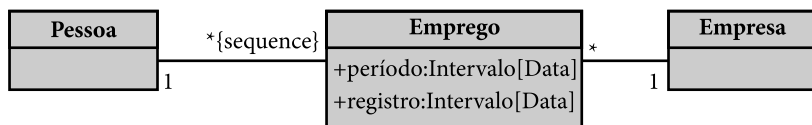


Figura 7.33:

Um modelo que implementa o padrão bitemporal.

Agora existem dois intervalos na classe *Emprego*, um para registrar o tempo no qual o emprego foi efetivo e outro para representar o período de tempo no qual essa informação foi registrada.

Cada vez que nosso conhecimento sobre o emprego de uma pessoa mudar, uma nova instância de *Emprego* é criada. O atributo *período* estabelece quando o emprego foi efetivo e o atributo *registro* estabelece o momento a partir do qual se passou a acreditar nisso. O limite superior do intervalo *registro* será mantido indefinido enquanto isso representar nosso conhecimento corrente, isto é, enquanto essa informação não tiver sido substituída por informação mais nova (possivelmente contraditória).

O padrão bitemporal também pode ser aplicado a atributos ou classe inteiras, da mesma forma que o padrão temporal.

7.13 Discussão

Um bom modelo conceitual produz uma estrutura organizada que é factível de gerar um banco de dados já normalizado. Ele incorpora regras estruturais que evitam que a informação seja representada de forma inconsistente; ele também simplifica o código que será gerado porque minimiza e organiza verificações de consistência, e várias verificações que são garantidas pelo próprio modelo não precisam ser realizadas pelo código.

O uso de padrões de projeto aplicáveis em situações onde eles são necessários simplifica o modelo conceitual e dá mais flexibilidade e qualidade. Ele é, portanto, uma ferramenta poderosa. Muitos outros padrões existem, e analistas podem criar seus próprios padrões. Apenas devem manter em mente que a criação de um padrão é justificada apenas quando seus benefícios justificam o esforço necessário para criá-los.

7.14 O processo visto aqui

	Concepção	Elaboração
Modelagem de negócio		
Requisitos		
Análise e design		Refinar o modelo conceitual: • Melhorar o modelo conceitual com a aplicação de padrões de análise.
Implementação		
Teste		
Gerenciamento de Projeto		

7.15 Questões

1. Na Figura 7.18, a classe *Transação* necessita de outra invariante. Uma transação não pode ser ligada a movimentações que estejam ligadas à mesma conta. Elabore essa invariante em OCL.
2. Aplique o padrão *conta/transação* às transações financeiras paralelas do exemplo mostrado nas Figuras 7.19 a 7.23. Como as contas podem ser pagas a prazo, poderá ser necessário usar movimentações mnemônicas. Defina o conjunto de instâncias de *Conta* e *Transação* que são necessários para descrevê-las.
3. Use o Google para encontrar exemplos de modelos conceituais que incluem classes com atributos. Analise essas classes e verifique se elas apresentam problemas de coesão. Proponha soluções alternativas a elas se necessário.
4. Um sistema para registro de árvores genealógicas permite que diferentes usuários verifiquem se eles têm pessoas coincidentes em suas árvores. Se for o caso, a pessoa compartilhada é marcada em cada uma das árvores como equivalente à outra. Assim, dois primos, por exemplo, que têm um avô em comum, podem ter um registro coincidente em suas árvores. Qual das três estratégias de junção de objetos seria melhor para lidar com este caso? Por quê? Elabore um modelo para representar pessoas nas árvores genealógicas usando este padrão.
5. Complete o diagrama da Figura 7.12 com informação sobre pedidos, itens, livros e compradores. Então use OCL para definir o atributo derivado *valor* para ambas as subclasses de *Desconto*.

Modelagem funcional com contratos OCL

TÓPICOS-CHAVE NESTE CAPÍTULO

- Precondições
- Retorno de consultas
- Pós-condições
- Exceções
- Contratos de operações de sistema

8.1 Introdução à modelagem funcional

Antes da modelagem funcional, o processo de análise na fase de Elaboração produziu dois artefatos importantes:

- O *modelo conceitual refinado* (Capítulos 6 e 7), o qual representa estaticamente a informação gerenciada pelo sistema.
- Os *casos de uso expandidos* ou *diagramas de sequência de sistema* (Capítulo 5), os quais mostram como usuários potenciais trocam informação com o sistema sem mencionar como a informação é processada internamente pelo sistema e sem mencionar a tecnologia de interface.

Quando diagramas de sequência de sistema são construídos, os comandos e consultas de sistema que devem ser implementados são identificados. Cada comando e consulta de sistema implica uma intenção ou objetivo do usuário; por exemplo, o usuário poderia fornecer informação sobre quais livros ele tenciona comprar ou poderia estar perguntando quanto um livro custa. Essa intenção é capturada pelos *contratos de comando de sistema* e *contratos de consulta de sistema*, genericamente referenciados como *contratos de operação de sistema*, os quais incorporam o modelo funcional do sistema.

Um *contrato de comando de sistema* pode ter três seções:

- *Precondições* (opcional): estabelecem o que é assumido como verdade pelo comando e que, portanto, não será verificado por ele.
- *Pós-condições* (obrigatório): estabelecem como o comando muda a informação existente se for executado com sucesso.
- *Exceções* (opcional): estabelecem quais condições que poderiam evitar que o comando tivesse sucesso as quais serão verificadas por ele.

No entanto, um *contrato de consulta de sistema* pode ter as seguintes seções:

- *Precondições* (opcional): estabelecem o que é assumido como verdadeiro pela consulta e, portanto, não será verificado por ela.
- *Resultados* (obrigatório): definem a informação que a consulta retorna.
- *Exceções* (opcional): estabelecem as condições que foram adicionadas pelo designer para evitar que o sistema realizasse a consulta sob determinadas situações.

Precondições existem nos dois tipos de contratos. Elas definem restrições que são complementares àquelas que já existem no modelo conceitual. São assumidas como verdadeiras pela operação. Isso significa que *precondições não* são testadas pela operação que é definida pelo contrato; elas devem ser testadas e garantidas *antes* que a operação seja executada.

Pós-condições existem apenas em contratos de comandos de sistema porque elas especificam *como a informação mantida pelo sistema é transformada*. Um cuidado especial deve ser tomado para não se confundir *pós-condições* com *resultados* de consultas. Pelo Princípio de Separação Comando-Consulta (Meyer, 1988), não é apropriado para um comando retornar qualquer resultado (exceto em casos especiais justificados). Consultas, no entanto, devem retornar algum resultado, mas não podem produzir qualquer mudança na informação existente. Portanto, consultas de sistema devem ter *resultados* e comandos de sistema devem ter *pós-condições*.

Em oposição às *precondições* que devem ser verdadeiras antes de a operação ser chamada, as *exceções* são situações que usualmente não podem ou que não se quer testar antes; as exceções são testadas pela própria operação que as contém. Exceções são eventos que, se ocorrerem, evitam que a operação seja concluída com sucesso.

Exceções em contratos de comandos e exceções em contratos de consultas têm propósitos distintos. Em contratos de comandos, exceções são usadas quando argumentos ou informação existente não satisfazem uma dada regra de negócio (por exemplo, tentar registrar um comprador que já está registrado).

Entretanto, em contratos de consultas, exceções têm um significado diferente. Como uma consulta não altera informação, não há argumentos inválidos que poderiam possivelmente evitar que uma consulta bem formada fosse realizada. Se um usuário fizer uma consulta ao sistema sobre um comprador que não existe, a consulta não é impedida de ser executada: ela deve responder que não existe tal comprador; esta é a resposta correta, não uma exceção. Exceções em consultas são usadas apenas se por algum motivo a equipe resolver que, para uma dada condição, a consulta *não deve* ser realizada. Usualmente, essas condições são relacionadas a questões de segurança ou outros assuntos tecnológicos, tais como interface multijanelas, concorrência, comunicação etc. Como esses aspectos usualmente não são tratados na modelagem funcional, exceções em contratos de consultas não são comuns aqui.

Normalmente, em um contrato de operação, seja comando ou consulta, uma exceção pode ser transformada em *precondição* e vice-versa. A decisão de projetar um aspecto

como condição ou exceção determina qual parte do sistema será responsável por testar o aspecto: a *operação chamada* ou a *operação que chama*. Usualmente, em sistemas grandes e multiusuário, as *condições de concorrência*¹ podem levar à escolha de evitar condições; entretanto, se o sistema não é alvo de concorrência complexa ou se um mecanismo de controle de concorrência consegue evitar que os dados mudem enquanto as transações ocorrem, então um projeto baseado em condições poderia ser a melhor escolha.

Para os exemplos deste capítulo, um refinamento parcial do modelo conceitual apresentado nos capítulos anteriores é usado (Figura 8.1), bem como uma versão *stateless* do diagrama de sequência de sistema para o caso de uso 01: *Pedir livros* (Figura 8.2).

O estereótipo `<<immutable>>` usado para alguns atributos e papéis na Figura 8.1 é explicado na Seção 8.7.2.

No modelo conceitual parcial da Figura 8.1, o conceito de um carrinho de compras foi adicionado porque o usuário pode iniciar comprando mesmo se ele ainda não tiver sido identificado. Assim, o carrinho de compras tem um código de identificação (porque a estratégia *stateless* está sendo adotada ele deve ser identificado de alguma maneira). Além disso, apesar do fato de que o carrinho de compras não é um conceito independente, ele deve estar conectado ao controlador, porque ele pode existir mesmo sem um comprador e mesmo sem seus itens (um carrinho de compras pode estar vazio).

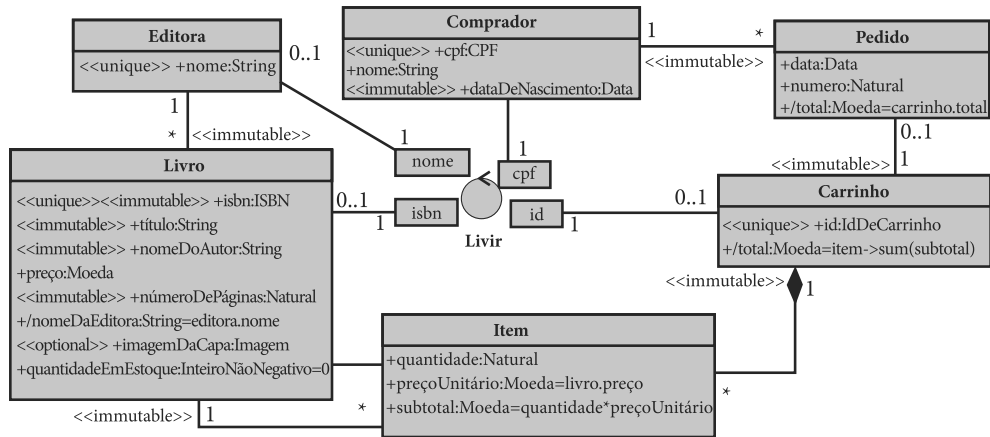
Na classe *Livro*, o atributo *quantidadeEmEstoque* poderia provavelmente ser um atributo derivado se o estoque fosse modelado com o padrão *Conta/Transação* (Capítulo 7). Mas aqui ele é mantido como um atributo simples².

O primeiro comando no diagrama de sequência de sistema cria um carrinho de compras e retorna seu identificador. Esta é uma exceção aceitável para o princípio de *Separação Comando-Consulta*, porque a interface necessita de uma referência para o carrinho de compras recém-criado para poder enviar mensagens para ele. Se a estratégia *stateful* estivesse sendo usada em vez da *stateless*, este retorno não seria necessário, porque o controlador poderia lembrar qual é o carrinho que acaba de ser criado.

Se o comprador não estiver logado, um fragmento chamado *login* é chamado (o qual pode ser especificado por um diagrama de sequência separado reusável). O fragmento retorna o identificador do comprador e substitui os dois fragmentos que foram mencionados na Figura 5.40: se o usuário tem uma conta, ele pode se logar; caso contrário, ele deve criar uma nova conta e se logar.

¹ Por exemplo, após um usuário solicitar um livro, enquanto ele está concluindo o pedido e providenciando pagamento, o único exemplar disponível acaba sendo vendido a outro usuário. Isso deve ser evitado com mecanismos de controle de concorrência, como será explicado no Capítulo 13.

² O tipo *InteiroNãoNegativo* é definido pelo conjunto {0, 1, 2, 3, ...} e se distingue do tipo *Natural* que exclui o zero.

**Figura 8.1**

Modelo conceitual de referência.

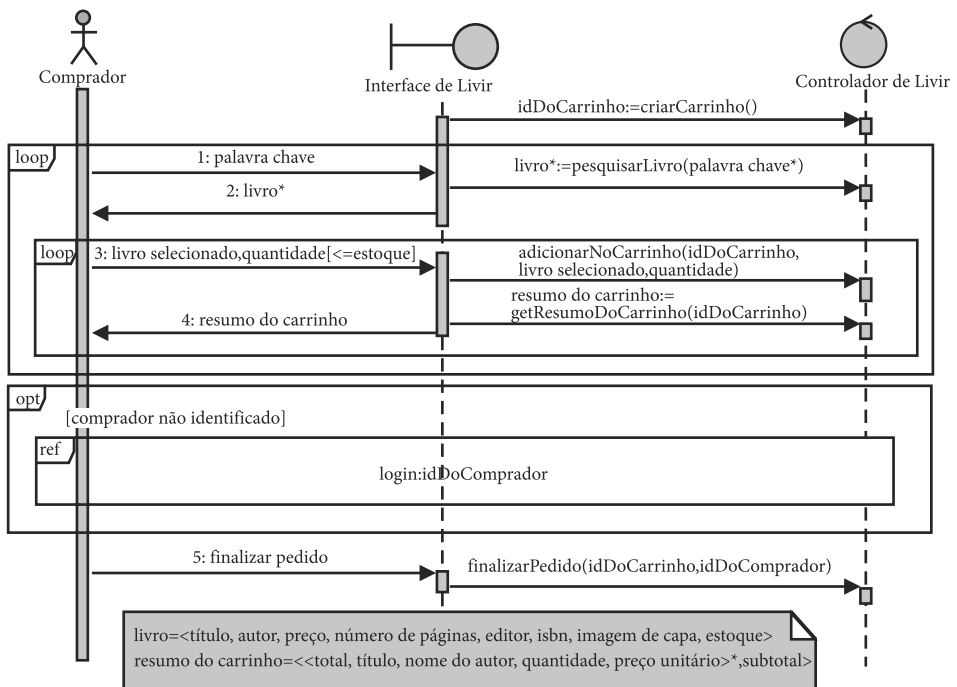
**Figura 8.2**

Diagrama de sequência de sistema de referência.

As operações (comandos e consultas) encontradas na Figura 8.2 são as seguintes (os argumentos mostrados no diagrama foram adaptados aqui para um estilo mais parecido com o usado nas linguagens de programação):

- *criarCarrinho():idDoCarrinho*: este é um comando que cria uma nova instância vazia de *Carrinho* e atribui um identificador a ela. Ele retorna o identificador.
- *pesquisarLivro(palavrasChave:Set<String>):OrderedSet<Tuple>*: esta é uma consulta que retorna um conjunto ordenado de tuplas que contém informação sobre os livros que são relevantes para a pesquisa. Cada tupla contém *título*, *nomes dos autores*, *preço*, *número de páginas*, *nome da editora*, *isbn*, *imagem de capa* e *quantidade em estoque* para um livro.
- *adicionarAoCarrinho(umIdDeCarrinho:IdDeCarrinho, umIsbn:ISBN, umaQuantidade:Natural)*: este é um comando que cria uma nova instância de *Item* e a liga ao carrinho e ao livro identificados como parâmetros.
- *getSumárioDoCarrinho(umIdDeCarrinho:IdDeCarrinho):OrderedSet<Tuple>*: esta é uma consulta que retorna um conjunto ordenado de tuplas com informações sobre o carrinho identificado pelo *id* de carrinho. A informação retornada, como estabelecido na nota da Figura 8.2, é *título*, *nome dos autores*, *quantidade*, *preço unitário* e *total*.
- *finalizarPedido(umIdDeCarrinho:IdDeCarrinho, umCpf:CPF)*: este é um comando que cria uma nova instância de *pedido* e a liga ao comprador e ao carrinho identificado pelos parâmetros.

8.2 Precondições

Precondições estabelecem o que deve ser verdadeiro quando uma consulta ou comando de sistema são executados. Por exemplo, considerando o modelo conceitual de referência da Figura 8.1, a operação *adicionarAoCarrinho(umIdDeCarrinho:IdDeCarrinho, umIsbn:ISBN, umaQuantidade: InteiroNãoNegativo)* pode assumir que o *id* do carrinho é válido porque ele foi obtido pela operação anterior *criarCarrinho():IdDeCarrinho*. Assim, uma precondição que estabelece que isso é verdadeiro poderia ser escrita assim:

```
Context Livir:adicionarAoCarrinho(umIdDeCarrinho:IdDeCarrinho,
    umIsbn:ISBN, umaQuantidade:Natural
pre:
    carrinho[umIdDeCarrinho]->notEmpty()
```

A expressão OCL anterior estabelece que no contexto do método *adicionarAoCarrinho*, que é um comando de sistema implementado pelo controlador *Livir*, há uma precondição que garante que existe de fato uma instância de *Carrinho* que é identificada pelo argumento *umIdDeCarrinho*.

Se a associação entre *Livir* e *Carrinho* na Figura 8.1 não fosse qualificada, então a precondição deveria ser escrita com a aplicação do *select* ao conjunto de carrinhos:

```
Context Livir::adicionarAoCarrinho(umIdDeCarrinho:IdDeCarrinho
    umIsbn:ISBN, umaQuantidade:Natural)
pre:
    carrinho->select(id=umIdDeCarrinho)->notEmpty()
```

Entretanto, quando uma associação é qualificada como na Figura 8.1, um valor de chave pode ser usada para acessar o elemento mapeado pela associação. Assim, como o comando *adicionarAoCarrinho* tem um argumento *umIdDeCarrinho*, a expressão *carrinho[umIdDeCarrinho]* produz o mesmo resultado que a expressão *carrinho->select(id=umIdDeCarrinho)*. Lembre que a última expressão também é uma abreviação de *self.carrinho->select(umCarrinho|umCarrinho.id=umIdDeCarrinho)*.

Para que sejam úteis para o desenvolvimento de software, as precondições devem ser expressas de forma cuidadosa. Elas devem refletir fatos que possam ser identificados no modelo conceitual previamente desenvolvido, ou o modelo terá de ser atualizado para refletir nova informação recém-descoberta. Isso justifica o uso de linguagens formais tais como OCL para escrever contratos (Warmer; Keppe, 1998) em vez da linguagem natural.

Duas famílias de precondições podem ser identificadas:

- *Garantia de parâmetros*: precondições que asseguram que os argumentos de um comando ou consulta correspondem a objetos válidos (por exemplo, “existe um carrinho de compras com o id passado como argumento”).
- *Restrições complementares*: precondições que restringem ainda mais o modelo conceitual quando um dado comando ou consulta estiver sendo executado, para assegurar que a informação esteja em um dado estado quando a operação for executada (por exemplo, “o carrinho tem pelo menos um item”).

8.2.1 Garantia de parâmetros

Há uma forma sistemática de saber se uma dada operação deve ter uma precondição de garantia de parâmetro (ou exceção). Essa técnica também se relaciona com o teste funcional das operações, como será visto no Capítulo 11. A equipe deve olhar cada parâmetro da operação e se perguntar se existem valores válidos e inválidos considerando o tipo do parâmetro.

Por exemplo, se o tipo do parâmetro for *ISBN*, e o comando for *adicionarAoCarrinho(..., umISBN, ...)*, então, considerando as regras de negócio, há um grupo de valores válidos (ISBNs que pertencem a um livro cadastrado) e um grupo de valores inválidos (ISBNs que são válidos mas não foram atribuídos a nenhum livro cadastrado no sistema).

No entanto, se o tipo do parâmetro for *String* e o comando for *adicionarAoCarrinho(..., umISBN:String, ...)*, então três classes devem ser consideradas: uma válida (a mesma de antes: ISBNs de livros já cadastrados), e duas classes inválidas consistindo dos ISBNs que são válidos, mas não pertencem a nenhum livro cadastrado (como antes), e uma de ISBNs malformados (porque o tipo *String* admite *qualquer* string, e não apenas ISBNs). Essa distinção entre ISBNs malformados e não cadastrados pode ser útil, por exemplo, para evitar erros de usuário relacionados com digitação incorreta quando este estiver entrando um valor de ISBN como parâmetro de busca.

Assim, considerando o tipo do parâmetro e identificando os grupos de parâmetros inválidos, a regra para encontrar precondições de garantia de parâmetros é a seguinte:

Para cada grupo de valores inválidos para um parâmetro de operação de sistema, deve ser definida uma precondição ou exceção.

Em relação às precondições de garantia de parâmetros, um cuidado especial deve ser tomado para não confundir precondições *semânticas* com simples verificações *sintáticas*. Verificação semântica requer que um banco de dados seja consultado: por exemplo, verificando se existe um livro com o ISBN dado. Entretanto, verificação sintática pode ser feita por outros mecanismos diferentes das precondições. Se um parâmetro tem tipo *Natural*, nenhuma consulta ao banco de dados é necessária para verificar se um dado argumento pertence ao tipo ou não. Assim, se um parâmetro é declarado como *x:Natural*, não é necessário escrever quaisquer precondições para garantir que *x* seja um número positivo: o próprio tipo já garante isso.

Outro exemplo é o ISBN, o qual tem uma regra de formação e pode ser verificado sintaticamente. Em vez de escrever precondições para verificar se um parâmetro é um ISBN bem formado, a equipe deveria simplesmente criar um tipo primitivo *ISBN* com um método de verificação, e usá-lo.

Mas que tal *NúmeroPrimo*? Poderia ser um tipo? A resposta é *sim*, porque para saber se um dado número é primo, uma verificação sintática é suficiente.

Tipos devem ser definidos na assinatura do comando. Se o tipo não existir, um tipo primitivo pode ser definido pela equipe. Por exemplo, a Figura 8.1 apresenta tipos primitivos tais como *CPF* e *IdDeCarrinho*, que podem incorporar não apenas os mecanismos de identificação e dígito verificador, mas também mecanismos de segurança, porque esses identificadores poderiam ser criptografados para evitar acesso não autorizado. Em ambos os casos, tipos primitivos poderiam ser definidos para lidar com a complexidade do mecanismo de verificação.

8.2.2 Restrições complementares

Uma restrição complementar consiste em assegurar-se que certas restrições mais fortes do que as do modelo conceitual são observadas enquanto um determinado comando é executado. Assim, se o modelo conceitual estabelece que um dado papel tem multiplicidade 0..1, por exemplo, uma restrição complementar poderia apenas dizer que quando um determinado comando estiver sendo executado, o papel poderá estar preenchido (1) ou não (0). Por exemplo, em geral, um *Carrinho* pode ter um *Pedido* associado a ele ou não (0..1). Mas o comando que finaliza um *Pedido* deve ter uma precondição que estabelece que o *Carrinho* não estava ainda ligado a qualquer pedido (o papel de *Carrinho* para *Pedido* deveria estar não preenchido: 0).

Assim, uma precondição nunca pode contradizer o modelo conceitual. Se o modelo conceitual estabelece 0..1, então nenhuma precondição poderá estabelecer multiplicidade

de 2 ou mais para o mesmo papel. Mas se o modelo conceitual estabelece 0..2, então a precondição pode restringir isso mais ainda, estabelecendo, por exemplo, 0..1 ou 1..2 etc.

Restrições complementares são comuns em designs que usam a estratégia *stateful*, porque, nesse caso, cada comando vai precisar de informações que foram deixadas na memória temporária por outros comandos, tais como, por exemplo, a identificação de um cliente, a qual é passada por um comando e usada por outros. Assim, esses outros comandos poderiam ter estabelecido como uma precondição que o cliente foi adequadamente identificado (possivelmente por uma associação temporária).

É possível identificar pelo menos três tipos de restrições complementares:

- Uma *afirmação específica* sobre uma instância ou sobre um conjunto de instâncias.
- Uma *afirmação existencial* sobre um conjunto de instâncias.
- Uma *afirmação universal* sobre um conjunto de instâncias.

Um exemplo de *afirmação específica* sobre uma instância, considerando novamente o modelo da Figura 8.1, poderia ser estabelecer que um comprador com CPF 000.000.000-00 tem *dataDeNascimento* igual a 14 de julho de 1970:

```
Context Livir: algumComando()
pre:
    comprador[000.000.000-00].dataDeNascimento=
        Date.getDate(1970,7,14)
```

Como OCL não define literais para o tipo *Date*, uma operação predefinida *Date.getDate* pode ser usada para definir uma data qualquer.

Um exemplo de *afirmação existencial* poderia ser dizer que existe pelo menos um comprador nascido em 14 de Julho de 1970 (não importa quem):

```
Context Livir:: algumComando()
pre:
    comprador->exists (umComprador |
        umComprador.dataDeNascimento=Date.getDate(1970,7,14))
```

Um exemplo de *afirmação universal* seria dizer que todos os compradores nasceram em 14 de Julho de 1970:

```
Context Livir:: algumComando()
pre:
    comprador->forall (umComprador |
        umComprador.dataDeNascimento=Date.getDate(1970,7,14))
```

Ambas as expressões *exists* e *forall* podem ser escritas de forma simplificada como *exists(dataDeNascimento = Date.getDate(1970,7,14))* e *forall(dataDeNascimento = Date.getDate(1970,7,14))*, mantendo o mesmo significado estabelecido anteriormente.

8.2.3 Garantia de precondição

Como as precondições não são testadas pelo comando que as declara, um mecanismo externo deve existir para garantir que elas serão verdadeiras antes que o comando seja chamado.

Esta garantia pode ser feita por uma chamada explícita que verifique se a condição é verdadeira antes de chamar o comando, ou por mecanismos de interface que evitam que o comando seja chamado com dados inválidos.

Por exemplo, em vez de permitir que o usuário digite qualquer *string* no campo ISBN, o usuário poderia ser obrigado a selecionar um item de uma lista de ISBNs válidos. Dessa forma, o parâmetro seria validado antes de o comando ser executado.

Usualmente, o fluxo de mensagens expresso no diagrama de sequência de sistema é uma forma de verificar se precondições são viáveis em vez de exceções. Por exemplo, na Figura 8.2 observe que o id do carrinho recebido pela operação *adicionarAoCarrinho* é válido porque ele foi retornado pelo comando *criarCarrinho*; o ISBN do livro é também válido, porque ele foi coletado de uma lista retornada pela consulta *pesquisarLivro*. Finalmente, a quantidade pode ser aceita como válida porque a interface pode compará-la com a quantidade disponível em estoque que foi retornada com outras informações sobre o livro³.

8.2.4 Precondições e exceções *versus* invariantes

Invariantes (Seção 6.7) são usadas no modelo conceitual para representar regras que são sempre válidas, independentemente de qualquer comando ou consulta. Precondições são usadas para regras que devem ser válidas apenas quando um dado comando ou consulta está sendo executado.

Quando uma invariante já existe para uma determinada situação, isso é equivalente à existência de uma exceção para qualquer operação. Por exemplo, se houver uma invariante que estabelece que um estudante pode se matricular apenas em disciplinas de seu próprio curso, então a invariante funciona como uma exceção para qualquer comando. Assim, comandos para matricular um estudante podem considerar que a exceção já existe na forma de uma invariante e se o estudante tentar se matricular em uma disciplina que não pertence ao seu curso, uma exceção vai ocorrer.

Entretanto, uma invariante não funciona como precondição. A existência da invariante não garante que o design não vai violá-la. Entretanto, um comando individualmente pode adicionar uma precondição que estabelece que a invariante deve ser garantidamente verdadeira antes de ele ser executado.

Mecanismos de teste devem verificar se as invariantes e precondições estão sendo respeitadas durante as atividades de teste do sistema. Se essas condições forem violadas a qualquer momento, elas podem dar origem a exceções. Entretanto, nesses casos, o designer deve revisar o design para corrigir este erro de forma que ele não ocorra mais. Quando o sistema for entregue ao usuário final, ele deve garantir que nenhuma invariante ou precondição seja violada a qualquer momento.

³ Lembre-se de que as condições de concorrência não foram ainda consideradas aqui e elas poderiam invalidar essa precondição se a concorrência for um problema.

8.3 Associações temporárias

Quando a estratégia *stateful* é usada no diagrama de sequência de sistema, é necessário para o controlador manter em memória certas informações que não são persistentes, mas que devem ser armazenadas durante a execução do caso de uso.

Alguns objetos, atributos, ou associações que não são persistentes podem ser definidos para indicar informação que é armazenada apenas na memória principal durante um caso de uso e descartada logo depois. Esses elementos podem ser estereotipados com `<<transient>>`.

Por exemplo, uma associação temporária poderia ser usada para indicar que o controlador deve guardar informação sobre quem é o comprador corrente, no modelo conceitual refinado, como mostrado na Figura 8.3.

Assim, uma precondição de um comando, por exemplo, poderia estabelecer que um comprador corrente já existe:

```
Context Livir::algunComando()
pre:
    compradorCorrente->notEmpty()
```

Atributos e classes temporárias, se necessário, podem ser definidos da mesma forma com este estereótipo. Em qualquer dos casos, isso significa que a peça de informação estereotipada não é persistente.

8.4 Retorno de consulta

Como mencionado antes, comandos de sistema mudam dados enquanto consultas apenas retornam dados para o usuário. Os contratos para consultas de sistema devem definir o que é retornado, e isso pode ser feito com OCL usando a cláusula *body*.

Expressões que representam precondições são sempre booleanas, mas expressões que representam o retorno de uma consulta podem ter outros tipos. Elas podem retornar *strings*, números, listas, tuplas ou mesmo estruturas mais complexas. Os exemplos a seguir são baseados no modelo da Figura 8.1. Inicialmente, uma consulta é definida para retornar o *total* de um dado carrinho:

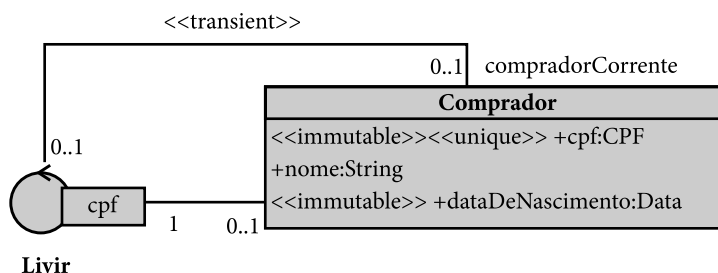


Figura 8.3

Uma associação temporária.


```
Context Livir::getTotalDoCarrinho(umIdDeCarrinho:IdDeCarrinho):Moeda
body:
    carrinho[umIdDeCarrinho].total
```

Consultas de sistema, bem como comandos de sistema, sempre têm o *Controlador* como seu contexto. Portanto, no exemplo anterior, *carrinho* é uma propriedade do controlador *Livir*: um papel de associação dele para a classe *Carrinho*.

A seguinte consulta retorna o nome e a data de nascimento de um dado comprador:

```
Context Livir::getNomeEDataDeNascimento(umCPF:CPF):Tuple
body:
    Tuple {
        nome=comprador[umCPF].nome,
        dataDeNascimento=comprador[umCPF].dataDeNascimento
    }
```

O construtor *Tuple* é uma das formas de representar *DTOs*⁴ em OCL; a tupla funciona como um *record* de Pascal com dois campos: *nome* e *dataDeNascimento*. Os valores dos campos são definidos pelas expressões após o “=“.

Para evitar repetir expressões como *comprador[umCPF]* ou mesmo expressões mais complexas, a cláusula *def* pode ser usada para definir uma constante que se refere à expressão original. Usando a cláusula *def*, o contrato poderia parecer com o seguinte:

```
Context Livir::getNomeEDataDeNascimento(umCPF:CPF)
def:
    umComprador=comprador[umCPF]
body:
    Tuple {
        nome=umComprador.nome,
        dataDeNascimento=umComprador.dataDeNascimento
    }
```

A expressão dentro da cláusula *def* indica que o identificador *umComprador* se refere ao comprador, cujo valor de CPF é igual ao parâmetro *umCPF* recebido pela consulta.

A seguinte expressão faz uma *projeção* no conjunto de compradores retornando os nomes de todos os compradores:

```
Context Livir::listaNomesDeCompradores():Set
body:
    comprador.nome
```

A expressão seguinte aplica um filtro e uma projeção, retornando os nomes de todos os compradores que têm menos do que 25 anos:

```
Context Livir::listaCompradoresJovens():Set
body:
    comprador->select(dataDeNascimento.addYear(25)>
        Date.getCurrent()).name
```

⁴ *Data Transfer Object* é um padrão de design que indica que objetos passados pelo controlador-fachada para a interface e vice-versa não podem ser instâncias de classes do domínio (aquelas do modelo conceitual). Eles devem ser dados puros, e não objetos de domínio com comportamento. Isso pode ser implementado com objetos que têm apenas atributos e seus respectivos *getters* e *setters* (se houverem), e nada mais. O tipo *record* em Pascal e *Tuple* em OCL são exemplos de DTOs.

A ideia no exemplo anterior é adicionar 25 anos à data de nascimento de cada comprador e comparar o resultado com a data corrente: se o comprador faz aniversário de 25 anos após a data corrente, então ele tem menos de 25 anos. OCL também tem operações para adicionar dias e meses às datas: *addDay* e *addMonth*, respectivamente. A expressão *Date.getCurrent* retorna a data corrente do sistema.

O último exemplo é uma consulta que retorna o resumo do carrinho, como mostrado na Figura 8.2.

```
Context Livir::getResumoDoCarrinho (umIdDeCarrinho:IdDeCarrinho):Tuple
def:
    umCarrinho=carrinho[umIdDeCarrinho]
body:
    Tuple {
        total=umCarrinho.total,
        itens=umCarrinho.item->collect(umItem|
            Tuple {
                título=umItem.livro.título,
                nomeDoAutor=umItem.livro.nomeDoAutor,
                quantidade=umItem.quantidade,
                preçoUnitário=umItem.preçoUnitário,
                subtotal=umItem.subtotal
            }
        )
    }
```

A expressão *collect* é uma forma de obter um conjunto cujos elementos são propriedades ou transformações das propriedades dos elementos de outro conjunto. A própria notação ponto (".") é uma forma abreviada de *collect*. Por exemplo, *comprador.nome* é equivalente a *comprador->collect(nome)*.

Quando possível, a notação ponto é preferível por ser mais curta. Mas no exemplo *getSumárioDoCarrinho* a necessidade de criar uma tupla em vez de acessar uma propriedade dos elementos de um conjunto impede o uso da notação ponto, visto que o operador *Tuple* é prefixado. Assim, a expressão *collect* tem de ser usada explicitamente nesse caso.

No exemplo, duas estruturas de tupla foram construídas: a externa contém o total do carrinho e o conjunto ordenado de seus itens; a interna contém para cada item a informação que é necessária como mencionado na Figura 8.2: *livro*, *título*, *nomes dos autores* etc.

8.5 Pós-condições

Pós-condições estabelecem o que muda na informação gerenciada pelo sistema após a execução de um comando. Pós-condições também devem ser cuidadosamente especificadas em termos que elas possam ser imediatamente reconhecidos na estrutura do modelo conceitual. Assim, embora contratos possam ser escritos em linguagem natural, eles devem ser cuidadosamente escritos de forma que possam ser imediatamente traduzidos em expressões OCL.

Uma pós-condição OCL é escrita no contexto de um comando de sistema usando-se a cláusula *post*:

```
Context Livir::algunComando()  
  post:  
    <expressão OCL>
```

Se houver mais do que uma pós-condição, então elas podem ser combinadas pelo uso do operador *and*⁵:

```
Context Livir::algunComando()  
  post:  
    <expressão OCL> and  
    <expressão OCL> and  
    ...  
    <expressão OCL>
```

Para classificar os tipos de pós-condições que podem ser usadas em um contrato, devemos tomar em conta que o modelo conceitual tem três elementos básicos, que são os conceitos, atributos e associação. Assim, considerando que instâncias de classes podem ser criadas e destruídas, ligações de associações podem ser adicionadas e removidas e valores de atributos podem ser modificados, apenas cinco tipos de pós-condições são possíveis:

- Alterar o valor de um atributo.
- Criar uma instância de uma classe.
- Adicionar uma ligação de associação.
- Destruir uma instância de uma classe.
- Remover uma ligação de associação.

Esses comandos são os mais fundamentais ou básicos que podem ser definidos para um sistema orientado a objetos. Eles não podem ser mais decompostos e todos os outros comandos são definidos como combinações deles.

Assim, um *comando básico* é um comando que realiza uma das cinco pós-condições listadas anteriormente. O significado e o comportamento de um comando básico são predefinidos.

Infelizmente, as linguagens de programação ainda não oferecem uma implementação padrão para tais comandos básicos. Implementar esses comandos algumas vezes é deixado como uma difícil tarefa para programadores, especialmente no caso de comandos que adicionam e removem ligações.

Os comandos básicos, conforme definidos nas seções seguintes, são efetivamente básicos no sentido de que eles realizam uma tarefa sem realizar certas verificações e consistência. Por exemplo, um comando que adiciona uma ligação não vai verificar o limite superior e inferior de ambos os papéis da associação que ele afeta. Este tipo de verificação de consistência é preferivelmente feito para um conjunto completo de comandos pertencentes ao mesmo contrato. A discussão sobre manter os objetos consistentes nos contratos é continuada na Seção 8.5.6.

⁵ O operador OCL *and* também pode ser usado para combinar precondições, exceções e invariantes.

8.5.1 Alteração do valor de um atributo

Um dos tipos de pós-condições consiste em indicar que o valor de um atributo foi alterado. Isso é feito em OCL usualmente da seguinte forma:

```
objeto.atributo=valor
```

Mas como OCL é uma linguagem declarativa, essa expressão não pode ser entendida como uma atribuição. Existem, de fato, três maneiras possíveis de ela ser verdadeira:

- *objeto.atributo* pode ser alterado para se tornar igual a *valor*. Essa é usualmente a interpretação *default*.
- *valor* pode ser alterado para se tornar igual a *objeto.atributo*.
- Tanto *valor* quanto *objeto.atributo* podem ser alterados para um terceiro valor e se tornarem iguais.

Portanto, em termos de geração de código, esse tipo de expressão pode produzir interpretações ambíguas. Cabot (2007) propõe uma semântica *default* para interpretar tais expressões de forma que a ambiguidade seja reduzida. Entretanto, outra abordagem é possível a qual não requer mudar a semântica original de OCL. A ideia é usar o predicado circunflexo (“^”) em vez do predicado de igualdade (“=”), conforme explicado a seguir.

Podemos indicar sem ambiguidade que um atributo mudou apenas declarando que uma mensagem básica relacionada com a mudança foi enviada ao objeto. Essa mensagem básica é predefinida e seu nome deveria iniciar com o prefixo “*set*” seguido do nome do atributo. O novo valor é passado como argumento.

A mensagem *setAtributo* pode ser enviada a uma instância da classe que contém o atributo. Por exemplo, se estamos interessados em mudar o preço de um dado livro, a seguinte expressão OCL poderia ser usada:

```
Context Livro::atualizaPreçoDoLivro(umISBN:ISBN,novoPreço:Moeda)
post:
    livro[umISBN].preço=novoPreço
```

Mas já foi visto que esta expressão pode ser interpretada de forma ambígua. A expressão a seguir expressa a mesma intenção, mas sem qualquer ambiguidade:

```
Context Livro::atualizaPreçoDoLivro(umISBN:ISBN,novoPreço:Moeda)
post:
    livro[umISBN]^setPreço(novoPreço)
```

A notação *circunflexo* significa que a mensagem à direita *foi enviada* ao objeto ou coleção de objetos na esquerda. Assim, ela corresponde a um predicado e o resultado da expressão anterior é booleano.

A expressão *livro[umISBN].preço=novoPreço* significa que dois valores são iguais, não importa como o resultado foi obtido. Mas *livro[umISBN]^setPreço(novoPreço)* estabelece sem ambiguidade que o atributo *preço* do objeto *livro[umISBN]* foi alterado para *novoPreço*.

Essa forma de escrita é ainda considerada como especificação declarativa, não imperativa. A expressão *livro[umISBN]^setPreço(novoPreço)* apenas estabelece que um objeto recebeu a mensagem; ela não diz quem enviou a mensagem. A decisão sobre qual objeto vai enviar essa mensagem para *livro[umISBN]* é deixada para a modelagem dinâmica (Capítulo 9).

8.5.2 Criação de instância

O comando que cria uma instância deve fazer apenas isso: criar uma instância de uma classe. Embora OCL não seja uma linguagem imperativa, ela tem um construtor para indicar que uma nova instância de uma classe foi criada. Isso é expresso como uma propriedade de um objeto e é referenciado como:

```
objeto.oclIsNew()
```

Seu significado, quando usado em uma pós-condição, é que o *objeto* foi criado durante o comando que está sendo especificado pela pós-condição. Essa propriedade pode ser usada em conjunto com outra que declara a classe do objeto:

```
objeto.isTypeOf(classe)
```

Entretanto, como poderia ser algo tedioso ficar declarando a criação de uma instância usando duas expressões, a definição de um predicado único que estabelece que um objeto foi criado como instância de uma determinada classe é sugerido. A expressão *objeto.isNewInstanceOf(classe)* é, portanto, definida como equivalente a:

```
objeto.oclIsNew() and objeto.oclIsTypeOf(classe)
```

Assim, uma pós-condição que declara que uma nova instância de *Livro* referenciada como *novoLivro*, foi criada, poderia ser definida como:

```
novoLivro.isNewInstanceOf(Livro)
```

Nesse caso, a notação circunflexo não é usada porque não se trata de uma mensagem enviada a uma instância de *Livro*; é a declaração de que uma instância *novoLivro* da classe *Livro* foi criada.

A mensagem *isNewInstanceOf* não afirma nada sobre os atributos e ligações da instância, mesmo as obrigatórias. Como antes, vai ser assumido que outras expressões vão inicializar os atributos e ligações obrigatórios, e que a consistência da nova instância será verificada para o contrato como um todo e não para cada comando básico individual.

Atributos com valores iniciais são considerados como automaticamente inicializados quando a instância é criada. Assim, seria redundante especificar pós-condições para alterar os atributos com valores iniciais, a não ser que valores diferentes do *default* sejam desejados.

Atributos derivados, no entanto, são calculados, e não podem ser diretamente modificados. Assim, eles não são inicializados.

Atributos estereotipados com *<<optional>>* podem ser mantidos indefinidos em tempo de criação; definir um valor para eles não é obrigatório.

Mas o que acontece com os outros atributos (normais e obrigatórios) no momento em que uma instância é criada? O comando básico, representado pelo predicado *isNewInstanceOf* simplesmente produz a instância; ele não inicializa atributos e ligações, exceto aqueles que foram definidos com valor inicial *default*. Se atributos e ligações obrigatórios não forem inicializados, a instância estará inconsistente; assim, o contrato poderia especificar pós-condições adicionais que assegurem que todos os atributos e ligações foram inicializados quando um objeto foi criado.

Até aqui vimos pós-condições para criar instâncias e alterar atributos. Isso nos permite combiná-las para criar um contrato para um comando CRUD que cria uma nova

instância de *Livro*. O comando de sistema *criarLivro* pode ter como um *rascunho inicial* de contrato o seguinte:

```
Context Livir::criarLivro (umISBN:ISBN,
    umTitulo,umNomeDeAutor:String, umPreço:Moeda,
    númeroDePáginas:Natural, umaImagemDeCapa:Imagem)
post:
    novoLivro.isNewInstanceOf(Livro) and
    novoLivro^setISBN(umISBN) and
    novoLivro^setTitulo(umTitulo) and
    novoLivro^setNomeDoAutor(umNomeDeAutor) and
    novoLivro^setPreço(umPreço) and
    novoLivro^setNúmeroDePáginas(númeroDePáginas) and
    novoLivro^setImagemDaCapa(umaImagemDeCapa)
```

Note que *quantidadeEmEstoque* é definido com um valor inicial e não precisa ser inicializado neste contrato.

O atributo *imagemDaCapa* é um atributo opcional e, por essa razão, é o único parâmetro que poderia aceitar o valor *null*; alternativamente, ele poderia ser suprimido da lista de parâmetros de *criarLivro*.

O atributo *nomeDaEditora* é derivado, e como é *read only* ele não pode ser definido no contrato (em vez disso, é a cláusula *derive* na própria classe que define seu valor para toda e qualquer instância). Entretanto, seu valor depende de uma ligação entre o livro e sua editora e a primeira versão desse contrato ainda não define como essa ligação vai ser adicionada. Essa discussão será feita na próxima seção.

Há mais uma preocupação aqui: se considerarmos que toda informação é acessível pela navegação nas ligações iniciando com a instância do controlador, então não seria recomendável declarar que uma instância foi criada, se ela não for imediatamente ligada a outra instância qualquer com um caminho de navegação até o controlador-fachada. A menos que outro mecanismo central de pesquisa permita encontrar instâncias desgarradas (não ligadas diretamente nem indiretamente ao controlador), a nova instância criada será inatingível. Portanto, a criação de uma instância deveria sempre acontecer em conjunto com a adição de uma ligação, o que será explicado na seção seguinte.

8.5.3 Adição de ligação

Conforme mencionado anteriormente neste capítulo, outro tipo de comando básico é o que estabelece que uma ligação entre dois objetos foi criada. Embora adições de ligações sejam limitadas pela multiplicidade dos papéis, isso só será verificado pelo contrato como um todo, e não para cada comando básico individual.

Usualmente, uma pós-condição OCL que indica que uma ligação foi criada poderia se parecer com o seguinte:

```
objeto.papel->includes(outroObjeto)
```

Mas novamente essa expressão pode ser interpretada de forma ambígua quando do momento de geração de código, porque existem pelo menos quatro maneiras de fazer com que ela se torne verdadeira:

- Incluindo *outroObjeto* em *objeto.papel* (a interpretação *default*).
- Atribuindo a *outroObjeto* um objeto que já esteja em *objeto.papel*.
- Atribuindo um terceiro objeto a *outroObjeto* e incluindo esse terceiro objeto em *objeto.papel*.
- Atribuindo o conjunto vazio a *outroObjeto*, porque independentemente do conteúdo de *objeto.papel* ele sempre inclui o conjunto vazio.

Usualmente, a primeira opção é o significado pretendido. Mas novamente sugerimos uma notação que não cause interpretações ambíguas e ao mesmo tempo é mais familiar para desenvolvedores de software. Novamente, a notação “^” e um comando básico são usados para indicar que uma ligação foi criada.

Existem vários dialetos para nomear comandos que modificam papéis de associação. Aqui, usamos o prefixo *add* seguido do nome de papel. Outra opção comum é usar o prefixo *set* como no caso de atributos. Entretanto, atributos têm seus valores alterados ou substituídos, enquanto ligações são adicionadas; assim, diferentes objetivos deveriam ter diferentes nomes.

Considerando a associação entre *Livro* e *Editora*, e considerando as instâncias *umLivro* e *umaEditora*, respectivamente, uma ligação entre essas instâncias poderia ser criada de duas formas diferentes. Primeiro, do ponto de vista do livro:

```
umLivro^addEditora(umaEditora)
```

Segundo, do ponto de vista da editora:

```
umaEditora^addLivro(umLivro)
```

Ambas as expressões são simétricas e produzem exatamente o mesmo resultado (uma ligação baseada na associação é criada entre as instâncias).

Associações com papéis obrigatórios, tais como a de *Livro* para *Editora*, usualmente requerem que suas ligações sejam criadas no mesmo contrato que cria o objeto que deve preencher o papel. Assim, uma ligação entre *Livro* e *Editora* será adicionada quando uma instância de *Livro* for criada, como mostrado adiante:

```
Context Livir::criarLivro(umISBN:ISBN,
    umTitulo, umNomeDeAutor:String, umPreço:Moeda,
    umNúmeroDePáginas: Natural, umNomeDeEditora:String,
    umaImagemDeCapa:Imagem)
post:
    novoLivro.isNewInstanceOf(Livro) and
    novoLivro^setISBN(umISBN) and
    novoLivro^setTitulo(umTitulo) and
    novoLivro^setNomeDoAutor(umNomeDeAutor) and
    novoLivro^setPreço(umPreço) and
    novoLivro^setNúmeroDePáginas(umNúmeroDePáginas) and
    novoLivro^setImagemDaCapa(umaImagemDeCapa) and
    novoLivro^addEditora(editora[umNomeDeEditora]) and
    self^addLivro(novoLivro)
```

No contrato anterior, duas ligações obrigatórias foram adicionadas ao novo livro: uma para sua editora, e outra dele para o controlador *Livir*. Lembre-se de que como *Livir* é o contexto do contrato, *self* representa a instância de *Livir*.

8.5.4 Destruição de instância

Embora a destruição de objetos seja rara em sistemas orientados a objetos, ela pode ocorrer algumas vezes. Usualmente, livros esgotados e compradores falecidos não são deletados dos registros: eles são simplesmente marcados como inativos ou movidos a um espaço alternativo. A informação não deve ser descartada para que registros confiáveis das atividades da empresa no passado sejam mantidos.

Entretanto, se uma editora ou qualquer outra instância de classe foi inserida no sistema de registros mais do que uma vez por engano, as instâncias podem ser unificadas se ambas já tiverem sido alteradas ou se nenhuma atividade foi registrada para a segunda instância, ela pode ser simplesmente deletada dos registros. Nesse caso, um objeto deve ser destruído.

Existem duas abordagens para indicar que uma instância foi destruída:

- *Explícita*: declara-se que o objeto foi destruído mandando-se a ele uma mensagem de destruição.
- *Implícita*: todas as ligações para o objeto são removidas de forma que ele se torna inacessível. Em linguagens de programação, é possível implementar *coletores de lixo* para remover da memória objetos que não são mais acessíveis.

Neste livro, é escolhida a abordagem explícita, porque ela deixa mais clara a verdadeira intenção do modelador. Um objeto que deve ser destruído, então, deve receber uma mensagem básica como a seguinte:

```
objeto^destroy()
```

O significado dessa expressão em uma pós-condição de comando de sistema é que o objeto referido foi destruído durante a execução do comando.

Assume-se que todas as ligações ao objeto destruído foram removidas também. Se qualquer objeto anteriormente ligado a ele se tornar inconsistente, então o contrato deve especificar o que acontece com aquele objeto (se ele é removido também ou se o papel é preenchido com um link para outro objeto).

8.5.5 Remoção de ligação

A remoção de uma ligação entre dois objetos é feita por um comando básico prefixado pela palavra *remove*⁶, seguido do nome de papel e recebendo como parâmetro o objeto, membro do papel, que deve ser removido. Por exemplo, para remover o último item de um dado carrinho, o seguinte contrato poderia ser feito:

⁶ Alguns dialetos poderia usar *unset*.


```
Context Livir::removeÚltimoItem(umIdDeCarrinho:IdDeCarrinho)
def: umCarrinho=carrinho[umIdDeCarrinho]
def: últimoItem=umCarrinho.item->last()
post:
    umCarrinho^removeItem(últimoItem)
```

Quando a multiplicidade do papel de destino da ligação a ser removida é 1 ou 0..1, é opcional informar o parâmetro, porque existe uma única ligação que pode ser possivelmente removida. Se em vez de remover a ligação do ponto de vista do carrinho isso fosse feito do ponto de vista do item, o resultado final seria o mesmo, mas o comando básico não precisaria de um parâmetro:

```
Context Livir::removeÚltimoItem(umIdDeCarrinho:IdDeCarrinho)
def: umCarrinho=carrinho[umIdDeCarrinho]
def: últimoItem=umCarrinho.item->last()
post:
    últimoItem^removeCarrinho()
```

Entretanto, ambos os contratos anteriores deixam o item inconsistente, porque ele tem uma associação obrigatória com *Carrinho* a qual não existe mais se o contrato atingir seu objetivo. O contrato deve considerar que como o papel de *Item* para *Carrinho* é obrigatório (1), a remoção da ligação implica a destruição do *Item* ou na sua ligação a outro *Carrinho*, o que deve ser obtido no mesmo contrato.

Se a escolha for deletar o item, então apenas o comando básico de destruição deveria ser usado no contrato, porque com ele todas as ligações para o item são consideradas automaticamente removidas:

```
Context Livir::removeÚltimoItem(umIdDeCarrinho:IdDeCarrinho)
def: umCarrinho=carrinho[umIdDeCarrinho]
def: últimoItem=umCarrinho.item->last()
post:
    últimoItem^destroy()
```

Tentar remover uma ligação não existente seria um erro de design e isso não deveria ser declarado em contratos bem formados, os quais são discutidos na seção seguinte.

8.5.6 Pós-condições bem formadas

Se assumirmos que os comandos básicos não verificam se os objetos e ligações são deixados em um estado consistente após serem executados, então o designer deve fazer a verificação quando ele estiver especificando os contratos de comando de sistema.

As verificações que precisam ser feitas podem ser resumidas conforme segue:

- Uma pós-condição que cria uma instância deve também ser acompanhada de pós-condições que estabelecem que todos os atributos da instância foram inicializados, com exceção de (1) atributos derivados (os quais são calculados); (2) atributos com valor inicial (que são predefinidos pela cláusula *init*, a não ser que um valor diferente do *default* seja desejado); e (3) atributos opcionais, os quais podem ser *null* (nesse caso a inicialização é opcional).

- Todas as ligações obrigatórias para uma instância recém-criada devem ser adicionadas.
- Mesmo que não tenha ligações obrigatórias, uma instância recém-criada deve ser ligada a pelo menos uma instância que tenha um caminho para o controlador ou seja ligada ao próprio controlador.
- Todas as ligações criadas ou removidas devem permanecer dentro dos limites inferior e superior de seus papéis de associação.
- Todas as invariantes afetadas pelas mudanças nos atributos, ligações ou instâncias devem permanecer verdadeiras.

O mecanismo de verificação de invariantes não garante que o designer esteja escrevendo um contrato que mantenha todas as invariantes verdadeiras. É responsabilidade do designer estar atento às invariantes e evitar causar exceções. Se uma invariante causa uma exceção que não é tratada pelo sistema, então existe certamente um problema de design.

Outra questão envolve atributos declarados com valores iniciais. Se a definição do valor inicial de um atributo usa informação que só pode ser obtida por meio da navegação pelas ligações do objeto, como ele poderia ser calculado antes de as ligações serem adicionadas? Deve-se considerar que a inicialização de atributos com valores iniciais será uma das últimas atividades no código de programação que implementa o contrato, ou seja, atributos com valores iniciais podem ser definidos apenas depois que todas as ligações forem adicionadas e a instância esteja consistente. Por exemplo, o *preçoUnitário* de um item na Figura 8.1 pode apenas ser inicializado depois que a ligação entre o item e o livro foi adicionada, porque o valor inicial depende dessa ligação. O designer do contrato não precisa se preocupar com isso, porque os contratos são declarativos, mas o programador deverá estar ciente disso quando for produzir o código.

8.5.7 Combinações de pós-condições

Usualmente, um comando de sistema terá várias pós-condições que podem ser unidas não só pelo operador *and*, como mencionado anteriormente, mas também pelo operador *or*, que estabelece que uma das pós-condições foi obtida, mas não necessariamente as duas:

```
post:
    <pós-condição 1> or
    <pós-condição 2>
```

O principal problema com este operador é que ele é *inerentemente ambíguo* em termos de geração de código e, assim, seu uso deveria ser evitado se expressões determinísticas forem possíveis.

Outro operador que pode ser usado para conectar expressões OCL é o operador *implies*, o qual tem o mesmo significado do operador de implicação da Lógica Booleana. *Implies* é usado quando uma pós-condição só deve ser obtida se certa condição for verdadeira. Por exemplo, o seguinte contrato adiciona um livro ao carrinho apenas se ele ainda não estiver lá:

```
Context Livir::adicionarAoCarrinho(umIdDeCarrinho:IdDeCarrinho,
    umISBN:ISBN, umaQuantidade:Natural)
def: umCarrinho=carrinho[umIdDeCarrinho]
def: umLivro=livro[umISBN]
def: umItem=umCarrinho.item->select(livro.isbn=umISBN)
pre:
    umCarrinho->notEmpty() and
    umLivro->notEmpty()
post:
    umItem->isEmpty() implies
    novoItem.isNewInstanceOf(Item) and
    umCarrinho^addItem(novoItem) and
    novoItem^setQuantidade(umaQuantidade) and
    novoItem^addLivro(umLivro) and
    novoItem^setPreçoUnitário(umLivro.preço)
```

O operador *implies* também pode ser trocado por uma expressão *if-then-endif*. Esta expressão é especialmente útil quando uma pós-condição alternativa deve ser obtida se a condição for falsa; nesse caso, a forma *if-then-else-endif* deveria ser usada. No caso do último exemplo, se o livro já está no pedido, a quantidade pedida pode ser adicionada à quantidade no item existente. O contrato a seguir mostra como usar este operador:

```
Context Livir::adicionarAoCarrinho(umIdDeCarrinho:IdDeCarrinho,
    umISBN:ISBN, umaQuantidade:Natural)
def: umCarrinho=carrinho[umIdDeCarrinho]
def: umLivro=livro[umISBN]
def: umItem=umCarrinho.item->select(livro.isbn=umISBN)
pre:
    umCarrinho->notEmpty() and
    umLivro->notEmpty()
post:
    if umItem->isEmpty() then
        novoItem.isNewInstanceOf(Item) and
        umCarrinho^addItem(novoItem) and
        novoItem^setQuantidade(umaQuantidade) and
        novoItem^addLivro(umLivro) and
        novoItem^setPreçoUnitário(umLivro.preço)
    else
        umItem^setQuantidade(umItem.quantidade@pre+umaQuantidade)
    endif
```

O significado do operador *@pre* será explicado na seção seguinte.

A vantagem do *implies* é que ele é mais curto. A vantagem do *if-then-endif* ou *if-then-else-endif* é que ele é mais familiar para programadores e inclui uma forma de especificar o que deve ser obtido em ambos os casos da expressão booleana, o que pode ser mais simples do que usar duas estruturas *implies*.

8.5.8 Valores prévios

Algumas vezes, uma pós-condição é construída a partir de valores que atributos ou papéis tinham antes de o comando ser executado. Esses valores prévios são referenciados pela função *@pre* para indicar que eles não correspondem ao valor do atributo *após* o comando ser executado (já que poderiam ser valores diferentes). Por exemplo, uma pós-condição que estabelecesse que uma quantidade em estoque para um dado livro foi incrementada pode ser expressa como:

```
Context Livir::adicionarAoEstoque(umISBN:ISBN,
    umaQuantidade:Natural)
def: umLivro=livro[umISBN]
post:
    umLivro^setQuantidadeEmEstoque(
        umLivro.quantidadeEmEstoque@pre+umaQuantidade
    )
```

Assim, porque *@pre* foi usada anteriormente? Porque, após o comando ser executado, a quantidade em estoque teria sido incrementada, e o argumento para o comando básico *setQuantidadeEmEstoque* deveria ser definido como a quantidade *prévia* somada à nova quantidade.

Lembre-se de que a OCL é declarativa, assim como a álgebra. A expressão $x = x + 1$ não é uma afirmação verdadeira na notação matemática, porque ela não pode ser verdadeira para nenhum valor de x . Mas quando se escreve $x := x + 1$ em uma linguagem de programação, isso significa o mesmo que $x = x@pre + 1$.

O operador *@pre* também pode ser usado sobre as ligações de um determinado papel. Por exemplo, para indicar o conjunto de itens que *estavam* ligados a um pedido antes de o comando de sistema ser executado, a segunda expressão poderia ser usada:

```
def: umPedido=...
post:
    umPedido.item@pre...
```

Outro exemplo com um uso mais intensivo deste operador é uma expressão que obtém o valor total derivado de um pedido antes que o comando de sistema fosse executado. Não apenas as quantidades e os preços dos itens podem ter mudado, mas a própria lista de itens pode estar diferente. Assim é necessário usar o operador *@pre* três vezes:

```
def: umPedido=...
post:
    umPedido.item@pre->sum(preçoUnitário@pre*quantidade@pre)...
```

Se, no entanto, o analista acredita que os valores dos atributos e ligações não foram alterados pelo comando de sistema, então a expressão *@pre* pode ser omitida, como no exemplo seguinte:

```
def: umPedido=...
post:
    umPedido.item->sum(preçoUnitário*quantidade)...
```

8.5.9 Pós-condições cobrindo coleções de objetos

É possível com uma única expressão OCL estabelecer que um conjunto inteiro de objetos foi alterado. Por exemplo, uma expressão que estabelece que o preço de todos os livros foi elevado em $x\%$ poderia ser escrita usando a expressão *forAll*:

```
Context Livir::elevaPreçoDosLivros(x:Percentual)
post:
    self.livro->forAll(umLivro|
        umLivro^setPreço(umLivro.preço@pre*(1+x))
    )
```

Se não houvesse necessidade de usar os valores prévios de preço na equação, ou seja, se todos os objetos vão ser atualizados com o mesmo preço, então seria possível abreviar ainda mais essa notação. Por exemplo, se o preço de todos os livros vai ser marcado em x reais, então a expressão seguinte poderia ser usada:

```
Context Livir::setPreçoDosLivros(x:Moeda)
post:
    self.livro->forAll(umLivro|
        umLivro^setPreço(x)
    )
```

Nesse caso, ainda existe uma forma mais abreviada:

```
Context Livir::setPreçoDosLivros(x:Moeda)
post:
    livro^setPreço(x)
```

Lembre-se de que em nossos exemplo, *livro* é um papel de uma associação do controlador para a classe *Livro*, e, portanto, essa expressão é equivalente a *self.livro*, ou seja, ela representa o conjunto de todos os livros que foram registrados no sistema. Como no caso da notação “*.*”, a notação “*^*” também pode ser usada sobre coleções.

8.5.10 Pós-condições e eventos do mundo real

O processo de criação de contratos está intimamente relacionado com a expansão do caso de uso e com o modelo conceitual. O modelo conceitual deve ser usado como uma referência a todo o momento, porque ele é a fonte de informação sobre a qual asserções são feitas sobre a informação.

Contratos deveriam ser sempre escritos com expressões que pudessem ser interpretadas em termos de elementos do modelo conceitual. Assim, asserções do tipo “os livros foram enviados” dificilmente poderiam ser pós-condições porque elas não representam informação contida no modelo conceitual; elas descrevem eventos do mundo real. Se a equipe quer expressar isso em um contrato, ela deve criar uma estrutura conceitual (atributo, papel ou conceito) para representá-la. Por exemplo, o fato de que livros foram enviados poderia ser representado pela criação de uma instância da classe *Envio* e pela sua ligação à instância atual de *Pedido*.

8.6 Exceções

Exceções em contratos são condições de falha que o designer não pode ou não quer evitar antes de executar o comando em questão.

Algumas vezes, condições identificadas como exceções podem ser convertidas em precondições. Se for possível transformar uma exceção em precondição, geralmente é recomendável fazer isso, porque é preferível evitar problemas o mais cedo possível. É preferível evitar que o usuário provoque um erro do que permitir que ele o faça e depois avisar que o erro ocorreu.

OCL não apresenta uma cláusula específica para gerenciar exceções, mas elas podem ser simuladas em contratos com pós-condições condicionais tais como:

```
post:
  if <condição> then
    <ativa exceção>
  else
    <todas as outras pós-condições, incluindo outras exceções>
  endif
```

Como poderia ficar difícil representar muitas exceções usando esta abordagem aninhada, propomos uma nova cláusula estereotipada, *exception*, a qual seria uma versão mais curta da expressão acima:

```
exception:
  <condição> implies <ativa exceção>
```

Assim, para um contrato indicar uma exceção, é suficiente identificar a condição que produz a exceção e o nome da exceção.

Um contrato apenas indica que as exceções podem ocorrer, mas seu tratamento deve ser definido apenas durante o design da camada de interface ou aplicação, porque se uma exceção é ativada pelo controlador-fachada, ela deve ser tratada pelo nível imediatamente acima: a camada de aplicação ou interface.

O seguinte exemplo mostra um contrato com uma exceção baseada nas Figuras 8.1 e 8.2:

```
Context Livir::finalizarPedido(umIdDeCarrinho:IdDeCarrinho,
  umCPF:CPF)
def: umComprador=comprador[umCPF]
def: umCarrinho=carrinho[umIdDeCarrinho]
pre:
  umComprador->notEmpty() and
  umCarrinho->notEmpty()
post:
  novoPedido.isNewInstanceOf(Pedido) and
  novoPedido^addCarrinho(umCarrinho) and
  novoPedido^addComprador(umComprador) and
  novoPedido^setData(Date.getCurrent()) and
  novoPedido^setNúmero(GeradorDeNúmeroDePedido.getNovo())
exception:
  umCarrinho.item->isEmpty() implies
    Exception.throw('Não se pode finalizar um pedido vazio')
```

Aqui, uma mensagem básica *throw* é usada para sinalizar a exceção para a classe chamada *Exception*, a qual não é uma classe original de OCL. Quando o código para este comando for gerado, a exceção deve ser sinalizada para o objeto que chamou o comando de sistema (possivelmente um objeto da camada de aplicação ou interface), e assume-se que nenhuma das pós-condições é obtida.

Uma exceção de comando de sistema deveria ser uma condição que não pode ser resolvida internamente pela implementação do comando de sistema e que requer que a execução do fluxo de controle seja retornada para a camada de interface, de forma que o usuário possa ser solicitado a tentar outras opções.

Exceções internas do software que um comando de sistema pode tratar internamente não são repassadas para a interface, portanto, elas simplesmente não são mencionadas nos contratos: elas serão tratadas quando os mecanismos internos de controle forem projetados e programados.

Também se assume que quando uma exceção é ativada, o comando não pode ser mais executado e nenhuma de suas pós-condições é obtida, porque os dados devem ser mantidos em um estado consistente mesmo quando uma exceção ocorre.

Como mencionado há pouco, algumas exceções podem ser convertidas em precondições se o designer conseguir distinguir um meio de garantir que a condição será verdadeira antes de chamar o comando. Assim, o contrato com exceção mostrado anteriormente poderia ser convertido no seguinte:

```
Context Livir::finalizarPedido(umIdDeCarrinho:IdDeCarrinho,
    umCPF:CPF)
def: umComprador=comprador[umCPF]
def: umCarrinho=carrinho[umIdDeCarrinho]
pre:
    umComprador->notEmpty() and
    umCarrinho->notEmpty() and
    umCarrinho.item->notEmpty()
post:
    novoPedido.isNewInstanceOf(Pedido) and
    novoPedido^addCarrinho(umCarrinho) and
    novoPedido^addComprador(umComprador) and
    novoPedido^setData(Date.getCurrent()) and
    novoPedido^setNumero(GeradorDeNumeroDePedido.getNovo())
```

Nesse caso, o comando assume que as operações prévias no diagrama de sequência de sistema já verificaram e garantiram que neste ponto o carrinho não está vazio; portanto, o comando não verificará isso novamente. Isso evita que o comando *finalizarPedido* realize verificações desnecessárias quando chamado e, assim sendo, isso simplifica seu código.

Apenas elementos conceituais (conceitos, atributos e associações) podem aparecer nas expressões dos contratos OCL. Esses elementos são necessariamente relacionados às regras de negócio do sistema sendo analisado. As exceções mencionadas aqui devem ser exceções relacionadas às regras de negócio e não exceções relacionadas a problemas

de hardware e comunicação. Exceções que podem ocorrer no armazenamento físico, comunicação ou dispositivos externos devem ser tratadas por mecanismos específicos nas camadas às quais estes elementos pertencem. Usualmente, usuários não ficam nem sabendo que elas ocorreram.

8.7 Contratos padrão para CRUD

As subseções seguintes apresentam modelos para contratos para operações CRUD típicas. Há três contratos para comandos de sistema e um para uma consulta de sistema. Os comandos e a consulta são realizados com a classe *Livro*, definida segundo o modelo da Figura 8.1. Contratos para operações CRUD são similares e podem ser considerados padrões.

8.7.1 Contrato para *criar*

O contrato para o comando *criar* usualmente deveria incluir a instanciação de um novo objeto, inicialização de seus atributos obrigatórios e adição de suas ligações obrigatórias. Como exemplo, o seguinte contrato poderia ser feito para criar (cadastrar) um novo livro:

```
Context Livro::criarLivro(umISBN:ISBN,
    umTitulo,umNomeDeAutor:String, umPreço:Moeda,
    umNúmeroDePáginas:Natural, umNomeDeEditora:String,
    umaImagemDeCapa:Imagem)
pre:
    editora[umNomeDeEditora]->notEmpty()
post:
    novoLivro.isNewInstanceOf(Livro) and
    novoLivro^setISBN(umISBN) and
    novoLivro^setTitulo(umTitulo) and
    novoLivro^setNomeDoAutor(umNomeDeAutor) and
    novoLivro^setPreço(umPreço) and
    novoLivro^setNúmeroDePáginas(umNúmeroDePáginas) and
    novoLivro^setImagemDaCapa(umaImagemDeCapa) and
    novoLivro^addEditora(editora[umNomeDeEditora]) and
    self^addLivro(novoLivro)
```

Este contrato assume que o nome da editora já foi verificado e é válido; caso contrário, esta condição deveria ser uma exceção em vez de precondição.

Como o atributo *isbn* já é estereotipado como <<unique>>, não é necessário estabelecer uma exceção relacionada ao caso de criação de um livro com um ISBN existente, porque esta exceção já seria ativada pela invariante de classe definida pelo estereótipo <<unique>>. O designer poderia querer tornar explícito no contrato que tal exceção existe; entretanto, seria redundante declarar uma exceção que já foi definida como invariante, e como a redundância é uma possível fonte de problemas, ela deveria ser evitada.

Entretanto, se a equipe quiser garantir que o argumento *umISBN* passado para o comando é realmente um ISBN que ainda não foi registrado, eles podem fazer isso pela adição de uma precondição para o contrato. Isso não invalida a exceção que poderia ser ativada pela invariante, mas isso garante que esta operação especificamente não vai ativar aquela exceção:

```
Context Livro::criarLivro (umISBN:ISBN,
    umTítulo,umNomeDeAutor:String, umPreço:Moeda,
    umNúmeroDePáginas:Natural, umNomeDeEditora:String,
    umaImagemDeCapa:Imagem)
pre:
    editora[umNomeDeEditora]->notEmpty() and
    livro[umISBN]->isEmpty()
post:
    novoLivro.isNewInstanceOf(Livro) and
    novoLivro^setISBN(umISBN) and
    novoLivro^setTítulo(umTítulo) and
    novoLivro^setNomeDoAutor(umNomeDeAutor) and
    novoLivro^setPreço(umPreço) and
    novoLivro^setNúmeroDePáginas(umNúmeroDePáginas) and
    novoLivro^setImagemDaCapa(umaImagemDeCapa) and
    novoLivro^addEditora(editora[umNomeDeEditora]) and
    self^addLivro(novoLivro)
```

8.7.2 Contrato para *atualizar*

O comando *atualizar* usualmente envolve apenas mudar os valores de alguns atributos, embora algumas vezes possa também envolver adicionar ou remover ligações. Antes de definir um contrato para atualizar um objeto, entretanto, o designer precisaria decidir quais atributos e ligações são imutáveis. O estereótipo *<<immutable>>* adicionado a alguns atributos e papéis de associação na Figura 8.1 declara que aquelas propriedades não podem mais mudar depois que o objeto é criado. Essa característica só é significativa quando se começa a considerar se objetos podem mudar ou não, e o momento da escrita dos contratos é justamente o momento de tomar essa característica em conta. No caso de um atributo imutável, seu valor pode ser definido quando uma instância for criada, mas não pode mais mudar depois disso. Um exemplo é o ISBN de um livro. Um papel imutável refere-se a uma ligação que é usualmente obrigatória e adicionada quando o objeto é criado. O objeto naquele papel não pode mais ser mudado. Por exemplo, um item é ligado a um carrinho quando é criado e não pode mudar depois para um carrinho diferente.

O comando *atualizar* não pode alterar atributos e ligações imutáveis. No caso de um *Livro* na Figura 8.1 apenas os seguintes atributos podem ser atualizados: *preço*, *imagemDaCapa* e *quantidadeEmEstoque*.

O contrato de atualização nesse caso, poderia se parecer com o seguinte:

```
Context Livir::criarLivro(umISBN:ISBN, umPreço:Moeda,
    umaImagemDeCapa:Imagem, umaQuantidade:Natural)
    def: umLivro=livro[umISBN]
    pre:
        umLivro->notEmpty()
    post:
        umLivro^setPreço(umPreço) and
        novoLivro^setImagemDaCapa(umaImagemDeCapa) and
        novoLivro^setQuantidadeEmEstoque(umaQuantidade)
```

O ISBN é usado apenas para localizar o livro. Mas como este valor é imutável, ele não pode ser atualizado. Se ele *pudesse* ser atualizado, então dois valores de ISBN deveriam ser passados como parâmetros: o antigo ISBN usado para localizar o livro e o novo ISBN que vai substituir o valor antigo.

Lembre-se de que *atributos únicos não são chaves primárias*, como explicado no Capítulo 6; eles são apenas atributos com valores que não se repetem em diferentes instâncias. Eles podem ser imutáveis ou não dependendo da natureza do conceito.

8.7.3 Contato para *deletar*

O comando que *deleta* um objeto deve considerar regras estruturais do modelo conceitual antes de decidir se um objeto pode ou não ser destruído.

No caso da Figura 8.1, por exemplo, uma instância de *Livro* não pode ser deletada sem uma decisão sobre o que acontece com as possíveis instâncias de *Item* eventualmente ligadas a ele, porque do ponto de vista de um *Item* uma ligação para um *Livro* é obrigatória.

Para realizar a remoção do objeto sem violar quaisquer regras estruturais, deve-se escolher uma das três abordagens a seguir:

- Assegurar por uma *precondição* que o objeto sendo deletado não vai deixar outros objetos inconsistentes em relação ao limite inferior de suas associações. Com essa abordagem, apenas livros que não tenham itens associados podem ser selecionados para serem deletados.
- Usar uma *exceção* para abortar o comando *delete* se ele for tentado sobre um objeto que vai deixar outros objetos inconsistentes. Se um usuário tentar deletar um livro que tenham itens ligados a ele, uma exceção vai ocorrer.
- Assegurar por uma *pós-condição* que todos os objetos que vão ficar inconsistentes após a deleção também serão deletados. Nesse caso, itens ligados a um livro vão ser deletados caso o livro o seja.

A terceira abordagem é usada quando se quer propagar o comando *delete* para todos os objetos que tenham associações obrigatórias com o objeto sendo deletado. Essa propagação é recursiva: a deleção vai continuar até que não reste mais nenhum objeto com

ligações obrigatórias para objetos que foram deletados⁷. Essa abordagem *não* pode ser usada sempre; existem situações nas quais a propagação da deleção vai contra regras de negócio. No caso de livros, por exemplo, deletar itens ligados a um livro que está sendo deletado poderia criar problemas com pedidos antigos que já foram registrados; isso não é aceitável. Entretanto, deletar itens quando um carrinho está sendo deletado é aceitável, porque se um carrinho é deletado espera-se que, pelas regras de negócio, toda a informação referente aos seus itens também seja deletada.

Um contrato possível que usa a abordagem de *precondição* poderia se parecer com o seguinte:

```
Context Livir::deletarLivro(umISBN:ISBN)
pre:
    livro[umISBN]->notEmpty() and
    livro[umISBN].item->isEmpty()
post:
    livro[umISBN]^destroy()
```

Como indicado anteriormente, a mensagem *destroy* remove a instância de *Livro* bem como suas eventuais ligações, as quais não precisam ser removidas uma por uma. As pre-condições garantem que o livro existe e que o livro não tem quaisquer itens ligados a ele. Se um dado livro tem itens, então o comando não pode ser chamado.

Um possível contrato que usa a abordagem de *exceção* poderia ser parecido com o seguinte:

```
Context Livir::deletarLivro(umISBN:ISBN)
pre:
    livro[umISBN]->notEmpty() and
post:
    livro[umISBN]^destroy()
exception:
    livro[umISBN].item->notEmpty implies
    Exception^throw('O livro não pode ser deletado porque
    cópias dele já foram vendidas')
```

Um possível contrato que usa a abordagem de pós-condição que propaga a deleção a todos os itens ligados a um carrinho poderia ser como o mostrado a seguir:

```
Context Livir::deletarCarrinho(umIdDeCarrinho:IdDeCarrinho)
pre:
    carrinho[umIdDeCarrinho]->notEmpty()
post:
    carrinho[umIdDeCarrinho].item^destroy() and
    carrinho[umIdDeCarrinho]^destroy
```

Essa pós-condição estabelece que, além do carrinho, todo item ligado a ele deve ser deletado também. Como a classe *Item* não tem associações com ligações obrigatórias vindas de outras classes, o processo de destruição pode parar aí (os livros ligados aos itens, por

⁷ De fato, no caso geral, o limite inferior de um papel é a informação que deve ser considerada. Por exemplo, se um papel tem multiplicidade 3..*, então a remoção de uma ligação quando existem apenas três já deixa o objeto inconsistente.

exemplo, não são destruídos). Caso contrário, seria necessário destruir outros objetos, e isso deveria ser estabelecido pelo contrato.

A abordagem de pós-condição deve ser escolhida sempre que for desejável propagar a deleção a todos os objetos que dependem do objeto sendo deletado. Isso é também útil quando uma composição é deletada; por definição, todos os componentes da composição também devem ser deletados, a não ser que tenham sido previamente removidos dela.

Entretanto, como mencionado anteriormente, há situações nas quais a propagação não é desejável. Por exemplo, remover um livro do catálogo não é possível se já existem pedidos registrados para ele. Nesse caso, as abordagens de precondition ou exceção deveriam ser escolhidas. A primeira requer que o ISBN de um livro que não pode ser deletado não seja jamais passado como argumento para o comando. Por exemplo, a lista de livros disponíveis para deleção na interface deveria conter apenas aqueles livros que podem realmente ser deletados. Se essa garantia não for possível ou desejável, a abordagem de exceção deve ser usada. Nesse caso, a interface deixa o usuário tentar deletar e, se não for possível, uma exceção é ativada.

Usualmente, sistemas de informação não permitem que a informação seja deletada. Um livro que foi removido do catálogo, por exemplo, não será deletado, mas marcado como *inativo*; se a informação sobre o livro for removida, então a informação sobre os pedidos antigos poderiam ficar inconsistentes em relação à realidade. Para implementar essa abordagem, uma solução simples é definir um atributo booleano *ativo* para a classe cujos membros podem ser desativados em vez de deletados. O valor *default* para este atributo é *true*, porque o objeto usualmente está ativo quando é criado. No exemplo, para a classe *Livro*, a solução é mostrada na Figura 8.4.

Nesse caso, um contrato para desativar um livro seria algo como:

```
Context Livir::desativarLivro(umISBN:ISBN)
pre:
    livro[umISBN]->notEmpty()
post:
    livro[umISBN]^setAtivo(false)
```

Livro
+ativo:Booleano=true <<immutable>><<unique>> +isbn:ISBN <<immutable>> +título:String <<immutable>> +nomeDoAutor:String +preço:Moeda <<immutable>> +númeroDePáginas:Natural + /nomeDaEditora:String=editora.nome <<optional>> +imagemDaCapa:Imagem +quantidadeEmEstoque:InteiroNãoNegativo=0

Figura 8.4

Uma variação da classe *Livro* cujas instâncias não podem ser deletadas, mas marcadas como inativas.

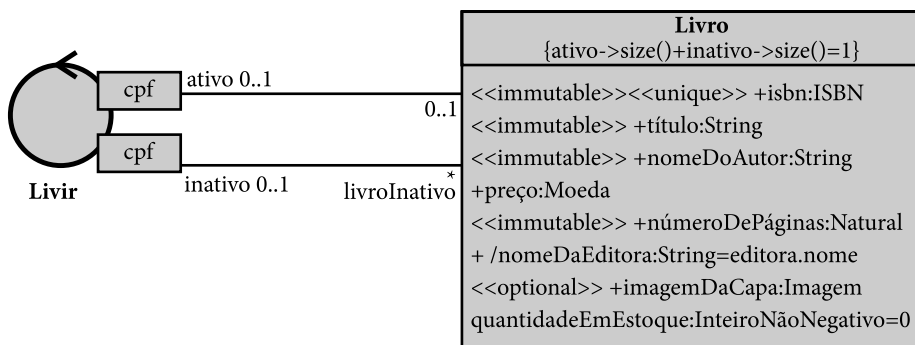


Figura 8.5

Forma alternativa de representar livros inativos com duas associações.

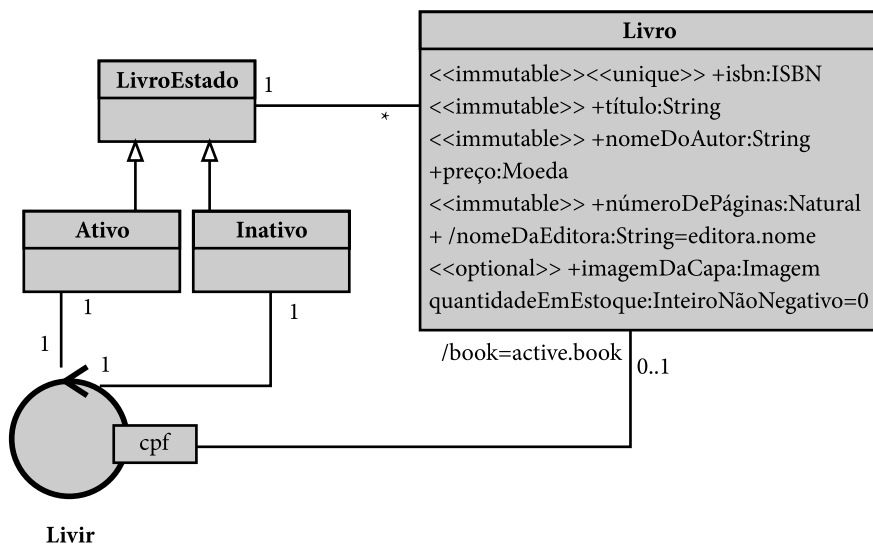


Figura 8.6

Uma forma alternativa de representar objetos inativos utilizando Padrão de design *Estado*

Não há necessidade de verificar se o livro tem itens nesse caso, porque nada será realmente deletado.

Outra abordagem seria mover os objetos inativos para um espaço diferente, por exemplo, usando uma nova associação do controlador que mantém objetos inativos separados dos ativos. A Figura 8.5 mostra um exemplo desta abordagem para a classe *Livro*.

Na Figura 8.5, há dois conjuntos de livros: *livro* (ativo) e *livroInativo*. A restrição na classe *Livro* garante que cada livro pertence a apenas um conjunto ou ao outro.

Ainda outra abordagem seria usar o padrão de design *Estado* para modelar os dois estados de um objeto, como mostrado na Figura 8.6. Novamente, há dois conjuntos de livros; o conjunto usual de livros (ativos) pode ser acessado diretamente por uma associação derivada, enquanto o conjunto de livros inativos na figura deve ser acessado pela consulta ao conjunto de livros ligados às instâncias de *Inativo* (entretanto, nada impede a criação de outra associação derivada para acessar livros inativos, se desejado). Este modelo é um pouco mais complexo visualmente, mas evita a necessidade de uma invariante na classe. Ele também poderia ser transformado em um padrão e um estereótipo poderia ser criado para ele.

8.7.4 Contrato para *consultar*

A *consulta* simples de CRUD retorna os dados disponíveis sobre uma instância de um dado conceito que é identificada por um de seus atributos únicos. Essas consultas não retornam dados que não estejam explicitamente na definição da classe. Consultas que retornam dados calculados a partir de um ou mais objetos são relatórios (casos de uso estereotipados com <<report>>). Operações de consulta não podem realizar cálculos.

Uma consulta simples para a classe *Livro* da Figura 8.1 poderia ser definida como segue:

```
Context Livir::consultarLivro(umISBN:ISBN):Tuple
  body:
    Tuple {
      isbn=umIsbn,
      título=livro[umISBN].título,
      nomeDoAutor=livro[umISBN].nomeDoAutor,
      preço=livro[umISBN].preço,
      númeroDePáginas=livro[umISBN].númeroDePáginas,
      nomeDaEditora=livro[umISBN].nomeDaEditora,
      imagemDaCapa=livro[umISBN].imagemDaCapa,
      quantidadeEmEstoque=livro[umISBN].quantidadeEmEstoque
    }
}
```

A consulta simples retorna uma tupla ou registro com os dados provenientes dos atributos do objeto selecionado pelo parâmetro da consulta.

8.8 Padrões de contrato para listar objetos

Frequentemente, é necessário criar listas de objetos onde um ou mais atributos são apresentados. Um primeiro exemplo desse tipo de contrato seria listar apenas o *título* de todos os livros cadastrados na livreria.

```
Context Livir::listarLivros():Set
  body:
    livro.título
```

Como o papel *livro* da controladora é um conjunto estrito na Figura 8.1, o resultado da expressão *livro.título* seria um conjunto também. Assim, se houver dois livros com o mesmo título, eles serão listados apenas uma vez, mesmo se os livros tiverem diferentes ISBNs.

Se uma lista com múltiplas colunas é desejada (por exemplo, autor e título), então é necessário retornar uma coleção de tuplas:

```
Context Livir::listarLivros():Set
body:
  livro->collect(umLivro|
    Tuple {
      nomeDoAutor=umLivro.nomeDoAutor,
      título=umLivro.título
    }
  )
```

Algumas vezes, é necessário aplicar um filtro à lista, por exemplo, retornando apenas o autor e título dos livros que não têm nenhum item ligado a eles (livros que nunca foram vendidos). Nesse caso, a função *select* pode ser aplicada ao conjunto antes de formar as tuplas:

```
Context Livir::listarLivrosNãoVendidos():Set
body:
  livro->select(
    item->isEmpty()
  )->collect(umLivro|
    Tuple {
      nomeDoAutor=umLivro.nomeDoAutor,
      título=umLivro.título
    }
  )
```

A maioria das consultas de listagem possivelmente terá apenas esses dois construtores: *select* (filtro) seguido de um *collect* (projeção). Mas consultas mais complexas podem ser criadas. Nesses casos, elas não necessariamente seguem o padrão de listagem (<<list>>) e são consideradas relatórios mais complexos (<<report>>).

8.9 Contratos relacionados a casos de uso

Um caso de uso expandido é uma cadeia de comandos e consultas que serão executadas em uma dada sequência. Usualmente, cada operação vai fornecer informações ou garantir precondições para outras operações. A melhor abordagem para escrever contratos para esta sequência de operações é seguir o fluxo do caso de uso que aparece no diagrama de sequência de sistema. Nessa sequência, o designer deve perguntar:

- Qual é o objetivo de cada operação?
- O que ela produz em termos de informação?
- O que ela espera que suas predecessoras tenham produzido?
- Que exceções poderiam ocorrer durante sua execução?

- Suas exceções podem ser reconfiguradas como precondições?
- Seus parâmetros incluem grupos de valores que são inválidos?
- As operações seguem algum padrão conhecido?

Quando responder a essas questões, o designer estará construindo contratos que permitem que os comandos e consultas sejam executados de uma maneira consistente no contexto da transação do caso de uso. Se for necessário adicionar novas consultas ou comandos ao diagrama de sequência com o objetivo e garantir certas precondições, então este é o momento correto para fazer isso.

Este capítulo mostrou vários exemplos relacionados ao diagrama de sequência de sistema da Figura 8.2. O leitor é encorajado a revisar aquela figura e perguntar a si mesmo as questões colocadas anteriormente para cada operação de forma a entender como elas foram respondidas durante o capítulo.

8.10 O processo visto aqui

	Concepção	Elaboração
Modelagem de negócio		
Requisitos		
Análise e Design		Escrever contratos de operação de sistema para comandos e consultas existentes nos diagramas de sequência de sistema: • Identificar precondições e exceções baseando-se nos parâmetros inválidos e restrições complementares. • Identificar pós-condições para comandos e retornos para consultas.
Implementação		
Teste		
Gerenciamento de Projeto		

8.11 Questões

1. Explique e apresente um exemplo de como uma exceção pode ser transformada em uma precondição e vice-versa.
2. Qual é a diferença entre exceções em contratos de comandos e em contratos de consultas?
3. Quais são as cinco possíveis pós-condições e quais os comandos básicos que definem cada uma delas? Como elas podem ser representadas em contratos OCL?
4. Produza o contrato para a consulta *pesquisarLivros* da Figura 8.2.

Design da camada de domínio

9

TÓPICOS-CHAVE NESTE CAPÍTULO

- Responsabilidade
- Visibilidade
- Acoplamento baixo
- Delegação
- Modelagem dinâmica
- Diagrama de classes de design

9.1 Introdução ao design da camada de domínio

Design de software em geral busca produzir uma solução para um problema que já foi suficientemente clarificado pela análise. Os aspectos estáticos do *problema* estão representados no modelo conceitual e os aspectos funcionais nos casos de uso expandidos, diagramas de sequência de sistema e contratos de operação de sistema. Agora é hora de fazer o design da *solução* de software (e possivelmente hardware) que implementa os aspectos lógicos e tecnológicos do sistema. Nesse sentido, *design é uma solução para um problema que foi modelado pela análise*.

Durante as iterações, após expandir os casos de uso, refinar o modelo conceitual e escrever contratos, a equipe pode conduzir atividades de design, as quais são divididas em dois grupos:

- *Design lógico* (este capítulo), que inclui os aspectos do problema que estão relacionados à lógica de negócio. Usualmente, o design lógico é representado no *diagrama de classes de design (DCD)*, o qual evolui a partir do modelo conceitual, e nos diagramas de interação, os quais mostram como os objetos trocam mensagens para realizar operações de sistema e obter as pós-condições dos contratos.
- *Design tecnológico*, que inclui os aspectos do problema que são inerentes à tecnologia usada: interface, armazenamento de dados, segurança, comunicação, tolerância a falhas etc. Atividades relacionadas ao design tecnológico são abordadas adiante:
 - *Design da camada de interface* (Capítulo 12), o qual consiste em projetar a interface com usuário e preferencialmente mantê-la desacoplada da camada de domínio.
 - *Design da camada de persistência* (Capítulo 13), o qual consiste na definição de um mecanismo de persistência para permitir que os dados sejam salvos e carregados, o que pode ser feito automaticamente, evitando que o designer tenha excesso de preocupação com esses aspectos.

O *design lógico* é também conhecido como *design da camada de domínio*. A camada de domínio corresponde ao conjunto de classes que realizam toda a transformação de dados e consultas. Outras camadas implementam aspectos tecnológicos do sistema: persistência, interface, comunicação, segurança etc. Elas são usualmente derivadas e dependentes da camada de domínio e sua utilidade é conectar a lógica pura do domínio com os aspectos físicos da computação (redes de comunicação, interfaces com humanos, dispositivos de armazenamento etc.).

O design da camada de domínio consiste basicamente de duas atividades que podem ser realizadas iterativamente:

- *Modelagem dinâmica*, que consiste em construir modelos de execução para os contratos de operação de sistema. Em sistemas orientados a objetos, tais modelos são usualmente representados por diagramas de interação, tais como os diagramas de comunicação e diagramas de sequência, ou mesmo por algoritmos puramente textuais.
- *Construção do DCD*, que consiste basicamente em adicionar ao modelo conceitual informações que não era possível ou desejável obter antes, tais como, por exemplo, a direção de navegação das associações e os métodos a serem implementados em cada classe. Esses aspectos podem ser efetivamente incluídos no design do software se a modelagem dinâmica for feita. Adicionalmente, a estrutura das classes de design pode ser um pouco diferente do modelo conceitual já que novas classes ou uma organização distinta das classes podem ser necessárias para realizar algumas funções que serão requeridas neste ponto.

O design lógico pode ser sistematicamente realizado se a equipe estiver de posse de dois artefatos:

- O diagrama de classes de design em evolução.
- O modelo funcional representado pelos contratos de operação de sistema.

As atividades de design lógico consistem em construir diagramas de interação para as operações de sistema encontradas nos diagramas de sequência de sistema (especialmente comandos de sistema, conforme explicado adiante), tomando em conta o DCD e o respectivo contrato de operação de sistema.

A modelagem dinâmica, como mencionado anteriormente, pode fazer uso de diagramas de comunicação ou sequência ou mesmo algoritmos. Cada uma das formas tem vantagens e desvantagens:

- *Algoritmos*, para programadores são mais fáceis de escrever, mas pode ser difícil de perceber claramente as conexões entre os objetos a partir de simples texto. Assim, a escolha por algoritmos para fazer modelagem dinâmica orientada a objetos pode aumentar o risco de obter classes com alto grau de acoplamento, o que produz um design ruim.
- *Diagramas de comunicação* são melhores do que algoritmos para propósitos de visualização e distribuição de responsabilidades, e eles são melhores do que os diagramas de sequência por fornecer uma visão explícita das linhas de visibilidade entre objetos.

Entretanto, esses diagramas podem ser difíceis de organizar e no caso de colaborações complexas eles podem se tornar mais difíceis de ler do que os diagramas de sequência.

- *Diagramas de sequência* são usualmente mais fáceis de ler do que os diagramas de comunicação, mas eles não mostram explicitamente as linhas de visibilidade entre os objetos. Se o designer não for vigilante, comunicações inválidas ou impossíveis podem acabar sendo incluídas no diagrama.

Este capítulo usa diagramas de comunicação para modelagem dinâmica para facilitar a visualização das linhas de visibilidade entre objetos. Em alguns casos, os diagramas de sequência equivalentes são mostrados para comparação.

O diagrama de classes de referência para os exemplos deste capítulo assume que estamos em tempo de design. Assim, associações poderão ser direcionadas e classes poderão ter métodos e atributos privados.

Inicialmente, o capítulo discute a responsabilidade de objetos, a qual é o conceito-chave para entender como os métodos se distribuem entre os objetos. Saber quais classes devem implementar quais responsabilidades permite fazer um design de classes altamente coeso.

Outro conceito-chave apresentado em detalhes é o de *visibilidade*, que estabelece que a comunicação entre os objetos deve ocorrer apenas onde existem linhas de visibilidade. Se a quantidade dessas linhas for minimizada, então o design obteve *baixo acoplamento* (Meyer, 1988).

Finalmente, o capítulo explica como transformar contratos em modelos dinâmicos pelo uso de distribuição de responsabilidades e visibilidade, bem como outros padrões de design, para produzir o melhor design possível.

9.2 Distribuição de responsabilidades de objetos

Distribuição de responsabilidades entre objetos tem a ver com a seguinte questão: *quais métodos devem ser implementados em quais classes?*

Muitos designers podem achar difícil implementar uma solução elegante para este problema quando eles tentam simplesmente adicionar métodos a um diagrama de classes. O uso dos diagramas de interação e padrões de projeto pode, entretanto, fornecer um meio mais eficiente e efetivo para *descobrir* os locais mais adequados para implementar cada método.

O DCD é construído a partir do modelo conceitual, o qual fornece o conjunto básico de classes, atributos e associações que são necessários. Então, é necessário desenvolver diagramas de interação para cada contrato de operação de sistema. Esses diagramas mostram objetos trocando mensagens para obter as pós-condições dos respectivos contratos. Existem técnicas que ajudam a construir esses diagramas de uma forma elegante. O uso de diagramas é melhor do que algoritmos, especialmente para designers iniciantes, porque com algoritmos o designer pode cair na tentação de concentrar todas as responsabilidades em uma única classe, enquanto os diagramas permitem uma melhor visualização da dis-

tribuição de responsabilidades entre os objetos. Um exemplo de tal situação é apresentado mais adiante neste capítulo.

Quando uma equipe constrói código orientado a objetos sem um método adequado para modelagem dinâmica (ou seja, simplesmente fazendo um diagrama de classes e adicionando métodos *ad hoc* a ele), existe um risco de que as responsabilidades sejam mal distribuídas entre as classes, e o resultado final poderá ser tão desestruturado quanto o antigo *código espaguete*. Usualmente, sistemas orientados a objetos são bem organizados em nível de classe, mas infelizmente o código escrito dentro dos métodos é frequentemente projetado de forma desorganizada.

Assim, para um sistema ser elegante, as responsabilidades devem ser bem distribuídas. Se um método sistemático não for usado, responsabilidades podem ficar concentradas na classe controladora, ou em classes que representam atores de negócio, tais como *Comprador* ou *Funcionário*. No final, classes como *Livro*, *Pedido* e *Pagamento* acabariam não tendo nenhum método relevante a não ser seus próprios *getters* e *setters* de atributos.

Quando uma ou duas classes fazem todo o trabalho e as outras são passivas, não há código orientado a objetos digno desse nome, mas uma estrutura concentradora. Seria preferível, talvez, fazer um bom *design estruturado* do que um design orientado a objetos malfeito.

No entanto, designers, algumas vezes, cometem o erro de acreditar que um sistema orientado a objetos é uma simulação do mundo real. Mas isso não é normalmente verdade. Um modelo orientado a objetos representa a *informação* sobre o mundo real e não as *coisas* em si. A diferença é sutil: métodos não correspondem a ações do mundo real, mas ao processamento interno de informação do sistema. É por isso que conceitos tais como *Livro*, que são passivos no mundo real, podem executar comandos em sistemas orientados a objetos.

Assim, fazer design de software orientado a objetos deve ser entendido como um método preciso, guiado por padrões aprendidos, e não simplesmente como o ato de criar classes e associar métodos *ad hoc* a elas.

Antes de explicar como distribuir responsabilidades entre objetos, uma taxionomia dessas responsabilidades precisa ser definida. Basicamente, existem dois grandes grupos de responsabilidades:

- A responsabilidade de *conhecer*, que corresponde às consultas sobre objetos.
- A responsabilidade de *atualizar*, que corresponde aos comandos realizados sobre objetos.

Ambos os grandes grupos se dividem em três subgrupos:

- Coisas que o objeto conhece ou atualiza sobre *si próprio* (seus atributos).
- Coisas que o objeto conhece ou atualiza sobre suas *vizinhanças* (suas ligações).
- Outras coisas que o objeto conhece ou atualiza que não são classificadas nos dois grupos prévios. Normalmente, essas coisas correspondem ao *conhecimento derivado*

(atributos e associações derivadas e outras consultas em geral) e *atividades coordenadas* (métodos delegados).

No caso das responsabilidades de *conhecer*, os três subgrupos poderiam ser caracterizados da seguinte maneira:

- *Coisas que o objeto conhece sobre si próprio*: isso é o equivalente a ser capaz de acessar os valores dos atributos do objeto. Tais responsabilidades são incorporadas às classes por meio de consultas básicas nomeadas com o prefixo *get* seguido do nome do atributo¹. Por exemplo, se a classe *Pessoa* tem um atributo *dataDeNascimento*, então o método *getDataDeNascimento* tem a responsabilidade de conhecer o valor deste atributo.
- *Coisas que o objeto conhece sobre suas vizinhanças*: isso é equivalente a ser capaz de acessar outros objetos diretamente ligados. Essa responsabilidade é incorporada nos métodos que obtêm o conjunto de objetos ligados por meio de um dado papel. Tais métodos são usualmente prefixados com *get* e seguidos do nome do papel (ou o nome da classe na falta do nome de papel). Por exemplo, se *Carro* é associado com *Pessoa* e o nome de papel para *Pessoa* é *motorista*, então o método *getMotorista* retorna o conjunto de pessoas que são motoristas de um dado carro.
- *Coisas que o objeto conhece de forma derivada*: isso é equivalente a informação que é composta ou calculada a partir de outras informações. Essa responsabilidade pode estar associada aos atributos derivados ou associações derivadas, quando os métodos são então prefixados por *get* seguido do nome do atributo ou papel de associação. Por exemplo, se *valorTotal* é o nome de um atributo da classe *Pedido*, então *getValorTotal* é um método que retorna este atributo derivado. Conhecimento derivado também inclui outras consultas sobre um objeto que não podem ser representadas como atributos ou associações derivadas, tais como, por exemplo, consultas parametrizadas; nesse caso, os nomes dos métodos podem variar.

No caso das responsabilidades relacionadas à *atualização* de Informação, os três subgrupos poderiam ser caracterizados como:

- *Coisas que o objeto atualiza em si próprio*: isso corresponde aos comandos básicos que atualizam um único atributo, o qual é prefixado com *set*², seguido do nome do atributo. Por exemplo, se a classe *Pessoa* tem um atributo *peso*, então *setPeso* é um método que incorpora a responsabilidade de atualizar este atributo.
- *Coisas que o objeto atualiza nas suas vizinhanças*: isso corresponde aos comandos básicos que adicionam e removem ligações entre objetos. Esses comandos são iden-

¹ Isso, porém, é válido para atributos normais pertencentes a classes do modelo conceitual. Durante a construção do DCD, novos atributos de design que representam estados internos e influenciam no comportamento de um objeto podem se tornar necessários e esses atributos não necessariamente poderão ser acessados por outros objetos: neste caso, eles não devem ter *getters* (eles são privados ou protegidos).

² Lembre-se de que nem todo atributo deve ter um *setter*. Atributos imutáveis não devem ter *setter*. Atributos derivados nunca têm *setter* e atributos privados também não deveriam ter *setters* públicos.

tificados respectivamente pelos prefixos *add* e *remove*³, seguidos do nome do papel da associação. Por exemplo, se a classe *Comprador* tem uma associação com *Reserva*, os métodos *addReserva* e *removeReserva* incorporam essa responsabilidade.

- *Atualizações que o objeto coordena*: isso corresponde a comandos que realizam múltiplas tarefas possivelmente em diferentes objetos. Esses comandos são conhecidos como *métodos delegados* e eles são cruciais para o design das classes, porque os métodos dos outros grupos são apenas comandos básicos com comportamento previamente conhecido; mas os métodos delegados precisam ser projetados. Por exemplo, se todos os livros em um determinado pedido precisam ser marcados como enviados, a coordenação das atividades de marcação deve ser feita pela própria instância de *Pedido*, porque ela é o objeto com o acesso mais imediato aos objetos que participam da operação.

A Tabela 9.1 resume os seis tipos de responsabilidade e os métodos tipicamente associados a cada tipo.

Não há um nome padrão para métodos delegados e consultas em geral, os quais são nomeados dependendo da operação que realizam. Usualmente, esses nomes não devem incluir o nome da classe na qual eles estão implementados. Por exemplo, se *Livir* implementa um comando *finalizarPedido* e esse comando delega responsabilidade para a classe *Pedido*, então, o nome do método implementado a classe *Pedido* deve ser simplesmente *finalizar* porque ele provavelmente seria invocado como *umPedido.finalizar()*.

Tabela 9.1 Responsabilidades de objetos		
	Conhecer	Atualizar
Sobre si próprio	Consulta sobre atributos: - <i>getAtributo</i>	Atualização de atributos: - <i>setAtributo</i>
Sobre suas vizinhanças	Consulta sobre papéis de associação: - <i>getPapel</i>	Atualizar ligações: - <i>addPapel</i> - <i>removePapel</i> - <i>replacePapel</i>
Outras	Consultas sobre atributos ou papéis de associações derivadas e consultas em geral: - <i>getAtributo</i> - <i>getPapel</i> - nomes variam (no caso de consultas em geral)	Métodos delegados: - nomes variam

³ Os prefixos *add* e *remove* poderiam ser usados aqui mesmo para associações para 1, porque conceitualmente elas são ligações como quaisquer outras. Entretanto, usualmente associações para 1 não implementam comandos como *add* e *remove*: elas implementam o comando *set* ou *replace* em lugar disso, o qual substitui a ligação obrigatória corrente por uma nova, deixando o papel sempre consistente.

9.3 Visibilidade

Para que dois objetos sejam capazes de trocar mensagens para realizar suas responsabilidades, é necessário que algum tipo de *visibilidade* exista entre eles. Existem quatro formas básicas de visibilidade entre objetos:

- *Por associação*: quando há uma ligação entre dois objetos definida por uma associação entre suas classes.
- *Por parâmetro*: se um objeto, quando estiver executando um método, recebe outro objeto como parâmetro.
- *Localmente declarada*: se um objeto quando estiver executando um método recebe outro objeto como retorno de uma consulta que ele enviou a um terceiro.
- *Global*: quando um objeto é visível por qualquer outro objeto.

Nenhuma das formas de visibilidade é simétrica. Isto é, se x tem visibilidade para y , isso não significa que y necessariamente tenha visibilidade para x . Visibilidade por associação até pode ser simétrica se a associação for bidirecional, mas esse não é sempre o caso.

9.3.1 Visibilidade por associação

Existe visibilidade por associação entre dois objetos apenas quando existir uma associação entre suas respectivas classes. O tipo de visibilidade permitida depende da multiplicidade de papel e outras características tais como o fato de a associação ser qualificada, ordenada etc.

Um papel de associação pode ser considerado uma coleção de instâncias, especialmente se seu limite superior for maior do que 1. Mas no caso de multiplicidade 1 ou 0..1, é possível interpretar o papel como um conjunto ou como um objeto individual. OCL, nesses casos, considera o papel tanto como um objeto quanto como uma coleção. Em relação à multiplicidade, então os seguintes tipos de visibilidade podem ser identificados:

- Se a multiplicidade é 1, então existe visibilidade direta para uma instância e para o conjunto que contém uma instância.
- Qualquer outra multiplicidade permite apenas visibilidade para um conjunto de instâncias⁴.

O DCD não é a única especificação que determina a multiplicidade de um papel. Ele é complementado pelas precondições dos contratos, as quais podem restringir os limites de multiplicidade mais ainda, como mostrado nas próximas seções.

⁴ Poder-se-ia considerar que um papel 0..1 seria preenchido com um único objeto ou com o valor *null*. Entretanto, isso não é coerente com a interpretação de outras multiplicidades, tais como, por exemplo, 0..2 ou *, onde 0 representa o conjunto vazio e não o valor *null*. Para evitar tal inconsistência, é preferível considerar que a multiplicidade 0..1 se refere a um conjunto que pode ser vazio ou conter um único elemento.

9.3.1.1 Visibilidade para um único objeto

A Figura 9.1 mostra a classe *Pagamento* que tem uma associação unidirecional para *Pedido* com multiplicidade 1. Nesse caso, qualquer instância de *Pagamento* tem visibilidade direta para uma instância de *Pedido*⁵.

Dessa forma, quando uma instância de *Pagamento* é representada em um diagrama de interação, como mostrado no diagrama da Figura 9.2, ela pode enviar mensagens diretamente para a instância de *Pedido* que está ligada a ela. Dessa forma, quando uma instância de *Pagamento* é representada em um diagrama de interação, como mostrado no diagrama da Figura 9.2, ela pode enviar mensagens diretamente para a instância de *Pedido* que está ligada a ela.

9.3.1.2 Visibilidade para múltiplos objetos

Valores de multiplicidade diferentes de 1, tais como *, 1..*, 0..1 ou 5, fornecem visibilidade para uma coleção⁶ de objetos, não apenas para uma única instância. Daqui por diante, todos esses casos serão tratados simplesmente como multiplicidade * ou associação *para muitos*.

A Figura 9.3 mostra a classe *Pedido* com uma associação unidirecional com multiplicidade * para a classe *Item*.

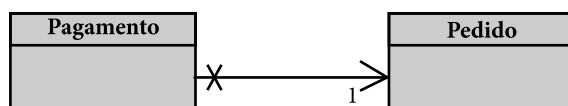


Figura 9.1

Papel de associação com multiplicidade 1.

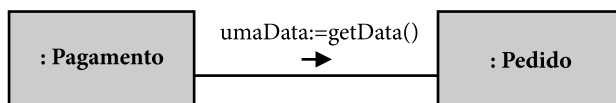


Figura 9.2

Diagrama de comunicação onde um pagamento tem visibilidade para um único pedido.

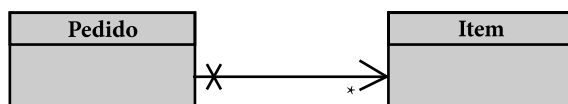


Figura 9.3

Papel de associação com multiplicidade *.

⁵ E adicionalmente para um conjunto que contém um pedido, mesmo se esta opção não for usada com frequência.

⁶ Um conjunto por default.

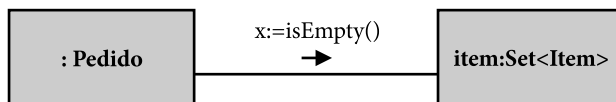

Figura 9.4

Diagrama de comunicação com uma mensagem sendo enviada a uma estrutura de conjunto.

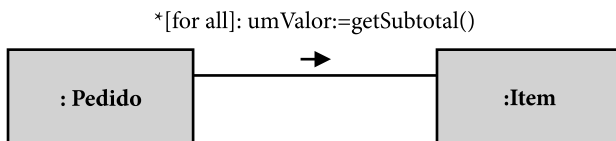

Figura 9.5

Diagrama de comunicação com uma mensagem sendo enviada a cada elemento do conjunto.

Nesse caso, uma instância de *Pedido* pode enviar uma mensagem a um conjunto de instâncias de *Item*. É possível enviar mensagens para a própria estrutura⁷, como mostrado na Figura 9.4. No exemplo, a mensagem verifica se o conjunto está vazio.

Mas é possível enviar uma mensagem iterativamente para cada elemento do conjunto, como mostrado na Figura 9.5. No exemplo, a mensagem solicita o subtotal de cada item.

Assim, as mensagens, em diagramas de comunicação, podem ser endereçadas à estrutura da coleção ou aos elementos da coleção individualmente. Quando uma mensagem é enviada à estrutura, o tipo do objeto que recebe a mensagem deve ser uma coleção, tal como *Set<Item>* na Figura 9.4. O fato de que a mensagem é iterativa e enviada a todos os elementos no conjunto é representada pelo “*” que precede a mensagem. A condição *[for all]* indica que todos os elementos do conjunto receberam a mensagem.

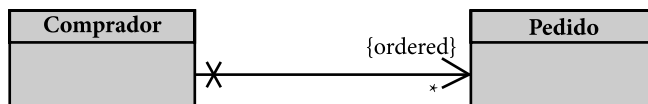
Um filtro poderia ser usado aqui se a mensagem for direcionada apenas para alguns elementos selecionados. Por exemplo, se a mensagem *getSubtotal* só deveria ter sido recebida pelos itens cuja quantidade está acima de 1, então ela poderia ser escrita como **[quantidade>1]: umValor:=getSubtotal()*.

9.3.1.3 Visibilidade por associação com papéis ordenados

Se o papel de associação é ordenado com o uso de restrições *{ordered}* ou *{sequence}*, a visibilidade para a coleção como um todo é mantida, mas adicionalmente visibilidade direta para elementos individuais indexados por sua posição é permitida.

A Figura 9.6 apresenta um diagrama de classes com uma associação unidirecional com um papel ordenado.

⁷ Neste texto, assume-se que as operações disponíveis para conjuntos e outras coleções são aquelas definidas pela OCL (Object Management Group, 2010). Entretanto, se diferentes linguagens ou pacotes forem usados, um conjunto diferente de operações pode estar disponível.

**Figura 9.6**

Papel de associação ordenado.

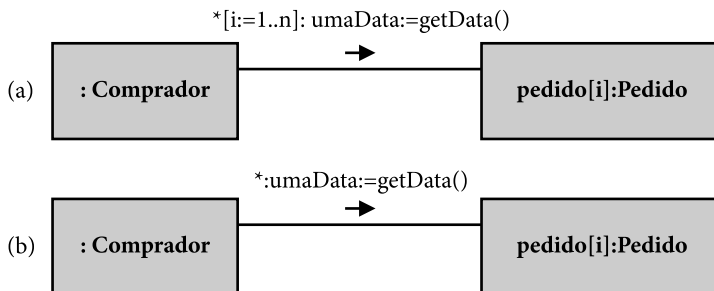
**Figura 9.7**

Diagrama de comunicação com uma mensagem sendo enviada a uma estrutura de conjunto ordenado.

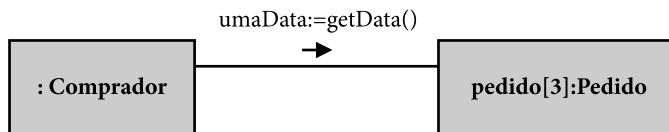
**Figura 9.8**

Diagrama de comunicação com uma mensagem sendo enviada a um elemento específico do conjunto ordenado.

Na Figura 9.7, a mensagem é enviada a cada elemento da coleção ordenada, como no caso da Figura 9.5.

Na parte A da Figura 9.7 é a forma explícita de representar uma mensagem iterativa enviada aos elementos de um conjunto ordenado ou sequência. A parte B é uma forma abreviada aceitável para fazer o mesmo. A forma **[for all]* usada para conjuntos também seria aceitável.

Na Figura 9.8, apenas o terceiro elemento do conjunto ordenado recebe a mensagem, e a visibilidade não é para múltiplos objetos como antes, mas para um único objeto. Isso é indicado no índice usado para representar o nome do objeto: *pedido[3]*. O valor entre colchetes pode ser uma constante, como no exemplo, ou uma variável conhecida pela instância de *Comprador*, ou mesmo uma expressão matemática que resulta em um inteiro.

Na Figura 9.9, apenas o último elemento da coleção ordenada recebe a mensagem.

9.3.1.4 Visibilidade por associação com qualificadores

Se a associação é qualificada como um mapeamento (apenas um elemento para cada valor do qualificador), existem pelo menos duas formas de visibilidade direta:

- Visibilidade para o conjunto de elementos como um todo (como se fosse um conjunto normal).
- Visibilidade para um único elemento se o objeto tiver uma chave para acessar o elemento no papel de associação.

A Figura 9.19 apresenta um diagrama de classes com uma associação qualificada de *Comprador* para *CartãoDeCrédito*.

A Figura 9.11 mostra que, nesse caso, uma instância de *Comprador* tem visibilidade para o conjunto de instâncias de *CartãoDeCrédito* que estão ligadas a ela.

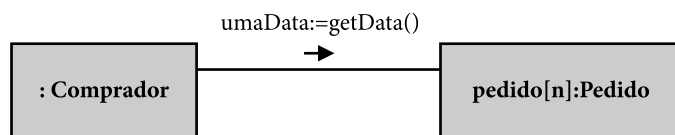


Figura 9.9

Diagrama de comunicação com uma mensagem sendo enviada ao último elemento de um conjunto ordenado.

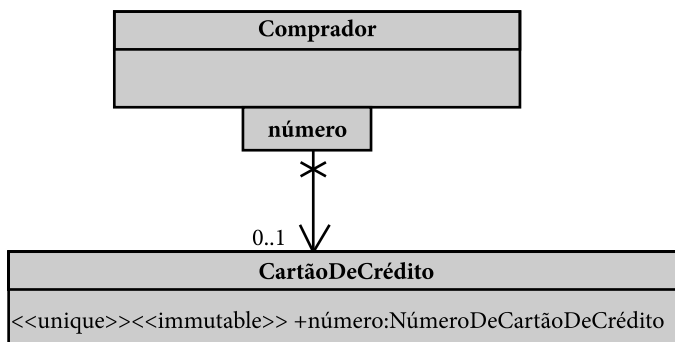


Figura 9.10

Diagrama de classes com uma associação qualificada definindo um mapeamento.

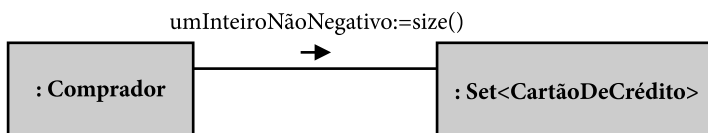


Figura 9.11

Diagrama de comunicação com uma mensagem sendo enviada para a estrutura de um mapeamento.

A Figura 9.12 mostra uma mensagem sendo enviada a cada um dos cartões de crédito no conjunto mapeado.

Finalmente, a Figura 9.13 mostra que se a instância de *Comprador* tem um valor de chave para o qualificador (*umNúmero* no exemplo), então ela tem visibilidade direta para o elemento do conjunto mapeado qualificado por aquele valor de chave.

No entanto, se a associação qualificada representa uma partição, como na Figura 9.14, um conjunto de valores é associado a cada qualificador.

Nesse caso, o conjunto inteiro pode ser acessado (todos os livros no exemplo da Figura 9.15).

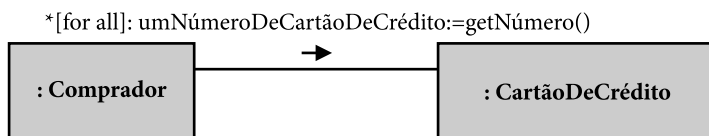


Figura 9.12

Diagrama de comunicação com uma mensagem sendo enviada a cada elemento do conjunto mapeado.

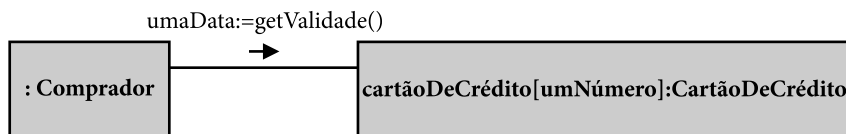


Figura 9.13

Diagrama de comunicação mostrando um objeto com visibilidade direta para um objeto qualificado em um conjunto mapeado.

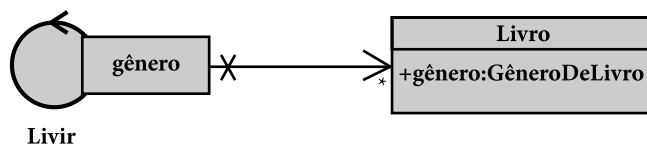


Figura 9.14

Diagrama de classes com uma associação qualificada definindo uma partição.

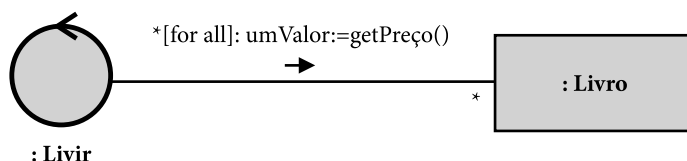
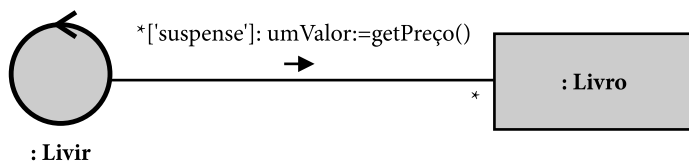
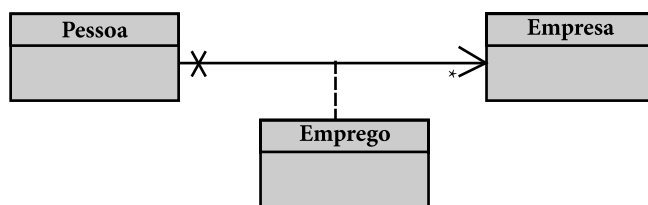


Figura 9.15

Diagrama de comunicação com uma mensagem sendo enviada para cada elemento do conjunto particionado.


Figura 9.16

Mensagem iterativa para cada elemento de uma partição de um conjunto qualificado.


Figura 9.17

Classe de Associação.

Mas o uso do qualificador permite o acesso a livros que pertencem a um dado sub-conjunto (livros de suspense, no exemplo da Figura 9.16).

Ambos os exemplos nas Figuras 9.15 e 9.16 mostram mensagens que são enviadas aos elementos dos conjuntos e não à estrutura dos conjuntos.

9.3.1.5 Visibilidade por associação com classes de associação

Como explicado na Seção 6.6.2, uma instância de classe de associação é criada e deletada quando links são adicionados e removidos das instâncias das classes participantes. Assim, se *Pessoa* tem uma associação unidirecional com *Empresa* e esta associação tem uma classe de associação *Emprego*, como mostrado na Figura 9.17, quando uma ligação entre instâncias de *Pessoa* e *Empresa* é adicionada, uma instância de *Emprego* é criada; e cada vez que uma ligação entre *Pessoa* e *Empresa* é removida, a respectiva instância de *Emprego* é deletada. Portanto, uma instância de *Pessoa* tem visibilidade para dois conjuntos⁸ com o mesmo tamanho: um com instâncias de *Empresa*, e outro com instâncias de *Emprego*. Uma instância de *Emprego* tem visibilidade apenas para uma única instância de *Empresa*.

Se a associação na Figura 9.17 fosse bidirecional, então uma instância de *Empresa* poderia também ter visibilidade para um conjunto de instâncias de *Pessoa* e um conjunto de instâncias de *Emprego*. Adicionalmente, qualquer instância de *Emprego* poderia ter visibilidade para uma instância de *Pessoa* e uma instância de *Empresa*.

⁸ Apesar disso, a associação provavelmente seria implementada não como dois conjuntos (ver Capítulo 10). Aqui, a visibilidade indica que dois conjuntos de objetos podem ser obtidos por uma instância; isso não significa que a associação vai ser implementada assim.

A Figura 9.18 representa a visibilidade *default* que uma instância de *Pessoa* tem para o conjunto de instâncias de *Empresa*. Isso corresponde à visibilidade *default* obtida para a multiplicidade $*$.

A Figura 9.19 representa a visibilidade que uma instância de *Pessoa* tem para um conjunto de instâncias de *Emprego*, a classe de associação.

A classe de associação também funciona como um mapeamento de forma bem similar à associação qualificada. O modelo da Figura 9.18 é equivalente a mapear um emprego para cada empresa na qual a pessoa trabalha. Assim, se uma instância de *Pessoa* tem visibilidade para uma instância específica de *Empresa*, ela pode encontrar seu emprego na empresa como ilustrado na Figura 9.20.

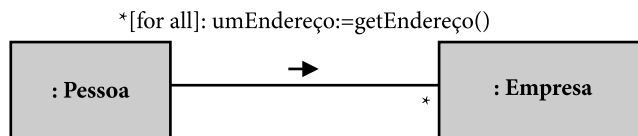


Figura 9.18

Uma mensagem sendo enviada aos elementos do conjunto ligado por uma associação com classe de associação.

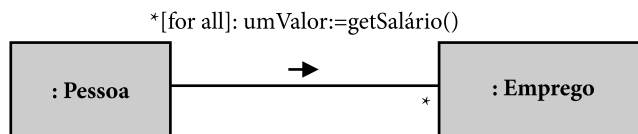


Figura 9.19

Uma mensagem sendo enviada aos elementos do conjunto de instâncias da classe de associação.

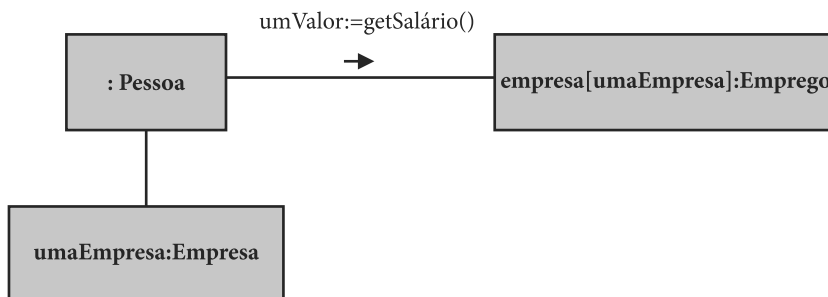
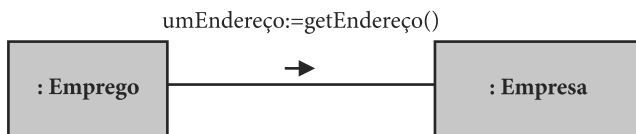
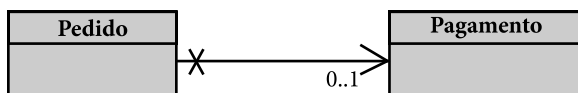


Figura 9.20

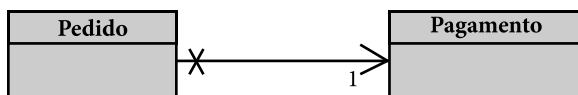
Uma mensagem sendo enviada a um elemento específico da classe de associação.


Figura 9.21

Visibilidade da instância da classe de associação para uma das instâncias das classes participantes.


Figura 9.22

Associação com papel opcional.


Figura 9.23

Como a visibilidade original da Figura 9.22 deve ser considerada para uma instância específica se o contrato tiver uma precondição mais restritiva.

Além disso, na Figura 9.20, pode-se ver que *umaEmpresa* deve ser uma instância de *Empresa* e que a instância de *Pessoa* deve ter algum tipo de visibilidade para ela para poder usá-la para acessar a respectiva instância de *Emprego*.

Finalmente, a Figura 9.21 mostra a visibilidade que instâncias da classe de associação têm para as instâncias das classes participantes: uma visibilidade que é estritamente para um.

9.3.1.6 A influência das precondições na visibilidade por associação

Quando uma precondição em um contrato estabelece uma multiplicidade que é mais restritiva do que a permitida no diagrama de classes, é a visibilidade estabelecida pela precondição que deve ser considerada válida durante aquela operação.

Por exemplo, considere o diagrama de classes da Figura 9.22, o qual define que um pedido pode ou não ter um pagamento ligado a ele.

Considere agora que um dado contrato de operação tenha uma precondição que estabeleça:

```

pre:
    umPedido.pagamento->notEmpty()
  
```

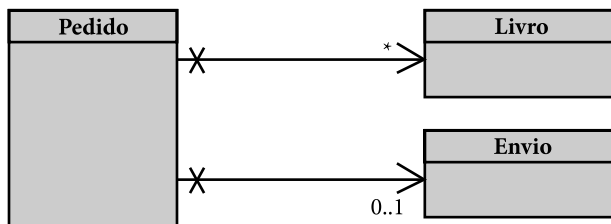
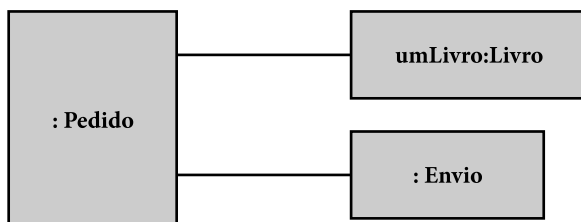
**Figura 9.24**

Diagrama de classes de referência.

**Figura 9.25**

Situação inicial na qual a instância de *Envio* não tem nenhuma visibilidade para a instância de *Livro*.

Então, no contexto dessa operação, e para a instância específica de *umPedido*, uma visibilidade mais restritiva deve ser considerada, como mostrado na Figura 9.23.

9.3.2 Visibilidade por parâmetro

Visibilidade por parâmetro é obtida quando um objeto *A*, executando um método, recebe um objeto *B* como parâmetro. Nesse caso, o objeto *A* pode se comunicar com *B* mesmo se suas classes não estiverem associadas.

Considere o exemplo da Figura 9.24, na qual a classe *Livro* não tem associação direta com a classe *Envio*. Portanto, nenhuma instância de *Livro* pode enviar uma mensagem diretamente a uma instância de *Envio* ou vice-versa.

Considere uma instância de *Pedido* que está ligada a uma instância de *Livro* e a uma instância de *envio* como mostra a Figura 9.25.

Agora considere que a instância de *Pedido* envia uma mensagem *registrarPeso* para a instância de *Envio*, passando como argumento uma referência para a instância de *Livro*, como mostra a Figura 9.26. Então, a instância de *Envio* adquire visibilidade por parâmetro para a instância de *Livro*.

No caso da visibilidade por parâmetro, deve ser assumido que o objeto que envia o argumento originalmente tinha algum tipo de visibilidade para o objeto que foi enviado.

Boas práticas de design exigem cuidado com a visibilidade por parâmetro. Este não é o mesmo tipo de acoplamento que é obtido com a visibilidade por associação. Com visibilidade por associação, as classes participantes já estão acopladas por uma associação semântica, isto é, elas já estão ligadas de qualquer maneira, porque o significado de uma está amarrado com o significado da outra (por exemplo, um pedido e seus itens). Mas, no caso da visibilidade por parâmetro, qualquer objeto pode ser passado como parâmetro para outro objeto, mesmo se não existir nenhuma relação entre eles.

Cada vez que uma classe declara métodos com parâmetros pertencentes a classes às quais ele não está associado, um novo (e possivelmente desnecessário) acoplamento está sendo criado. Alto acoplamento significa design ruim. Assim, é conveniente evitar isso sempre que possível.

O melhor uso para a visibilidade por parâmetro é receber e reenviar objetos até que eles atinjam as operações básicas ou terminais. Na prática, o que acontece é que certos objetos serão passados a outros objetos em uma cadeia até que ela alcance algum objeto que deve ser ligado ou desligado deles, como no exemplo abaixo.

Inicialmente, considere o diagrama de classes da Figura 9.27, com as classes *Livir* (o controlador-fachada), *Livro*, *Comprador* e *Reserva*.

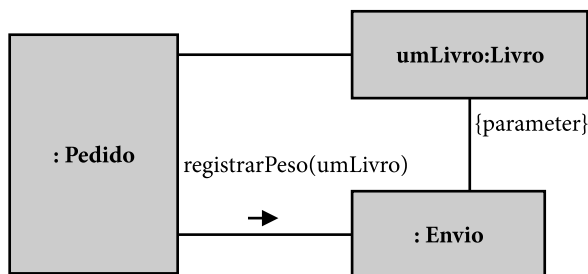


Figura 9.26

A instância de *Envio* adquire visibilidade por parâmetro para *umLivro*.

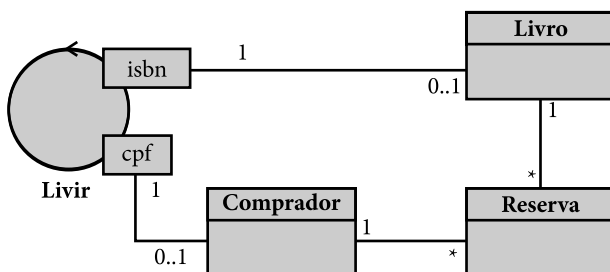
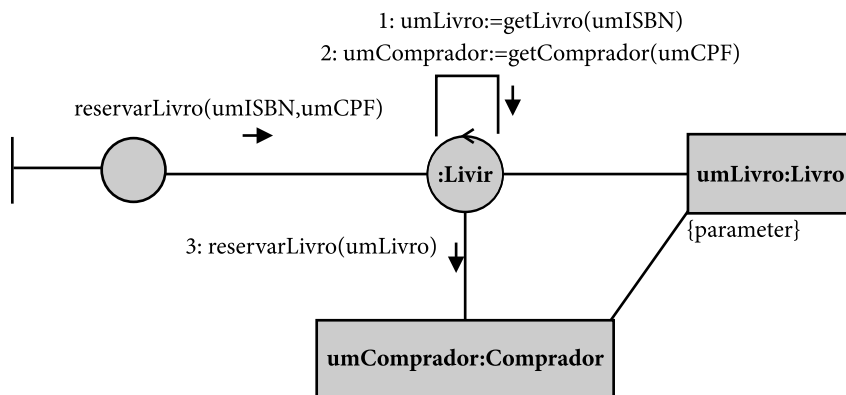
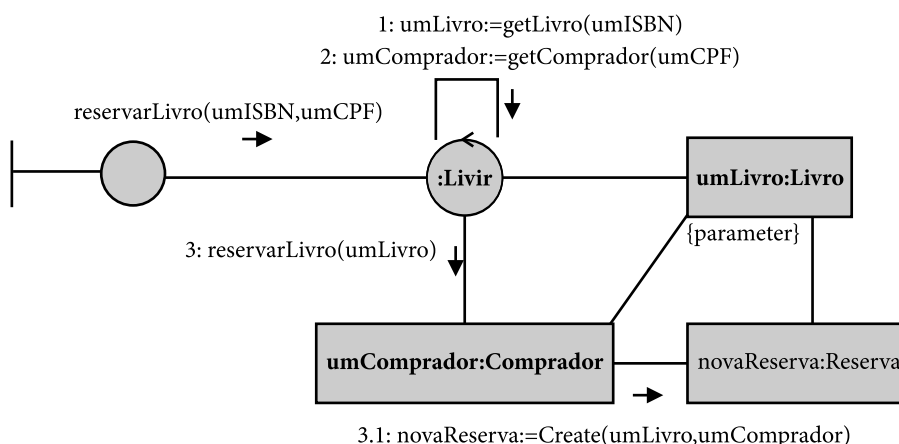


Figura 9.27

Diagrama de classes de referência.

**Figura 9.28**

Criação de visibilidade por parâmetro.

**Figura 9.29**Um diagrama de comunicação no qual uma nova instância de *Reserva* é criada.

A Figura 9.28 apresenta uma colaboração na qual instâncias dessas classes cooperam para realizar o comando de sistema *reservarLivro*, o qual adiciona uma reserva de um dado livro para um dado comprador. O comando de sistema, implementado no controlador, é chamado pela camada de interface, a qual passa como argumentos o ISBN do livro e o CPF do comprador. O controlador, inicialmente, ganha visibilidade para a instância específica de *Livro* (mensagem rotulada com “1”) ao enviar para si próprio a mensagem de consulta sobre o papel *livro*: a mensagem *getLivro(umISBN)*. Então ele ganha visibilidade para a instância de *Comprador* (mensagem rotulada com “2”) de uma forma similar. As instâncias retornadas pelas mensagens são atribuídas às variáveis locais *umLivro* e *umComprador*. Finalmente, o controlador delega para a instância de *Comprador* a realização da reserva do livro (mensagem rotulada com “3”).

A mensagem rotulada com “3”, *reservarLivro(umLivro)*, é enviada do controlador para o comprador cujo CPF é *umCPF*. O controlador não deveria ser responsável por criar a reserva porque ele não tem associação com *Reserva*, e assim ele delega a responsabilidade para *umComprador*. Neste ponto, *umComprador* adquire *visibilidade por parâmetro* para *umLivro*.

Nesse ponto, *umComprador* pode começar a realizar suas responsabilidades. Ele cria uma nova instância de *Reserva* e liga-a a si próprio e a *umLivro*. A Figura 9.29 mostra a mensagem construtora *Create* sendo enviada para criar uma reserva com seus parâmetros de inicialização: *umLivro* e *umComprador*. Essa mensagem é rotulada com 3.1, porque ela é chamada dentro da implementação da mensagem 3, a qual foi recebida por *umComprador*.

Como uma última observação sobre este exemplo, tente imaginar o que aconteceria se em vez de passar a instância de *Livro* ao comprador, o controlador passasse seu código de ISBN conforme recebido da interface. Quando a instância de *Reserva* for adicionar o livro às suas ligações, ela deveria obter a instância, porque ela não pode apenas adicionar o código ISBN. Como a instância de *Reserva* poderia pesquisar por uma instância de *Livro* se ela não tem acesso ao conjunto de todos os livros? Ela teria de mandar uma mensagem ao controlador, pedindo pelo livro. Isso não é um design de boa qualidade, porque o fluxo de mensagens, nesse caso, vai para trás, de volta para o controlador, sem uma boa razão. No exemplo mostrado na Figura 9.29, o controlador obtém o livro assim que ele recebe a mensagem original. Não há motivo para atrasar essa obtenção e reverter o fluxo de controle mais tarde.

9.3.3 Visibilidade localmente declarada

Outra forma de visibilidade ocorre quando um objeto envia uma consulta a outro e recebe como retorno um terceiro objeto, como será mostrado na Figura 9.31. A colaboração é baseada no diagrama de classes da Figura 9.30.

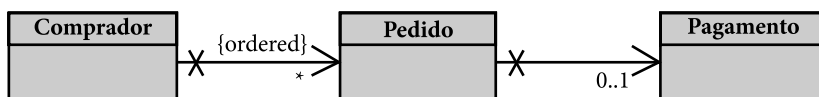


Figura 9.30

Diagrama de classes de referência.

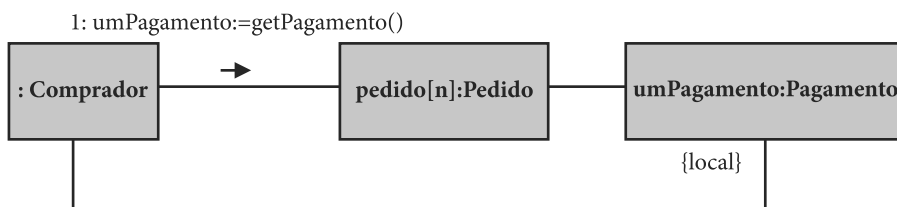
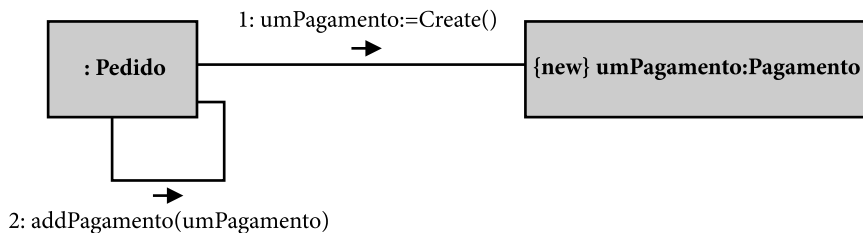


Figura 9.31

Um exemplo de criação de visibilidade local.

**Figura 9.32**

Um exemplo de visibilidade local *nova* (*new*) sendo transformada em visibilidade por associação.

No exemplo, uma instância de *Comprador* envia uma mensagem *getPagamento* a uma instância de *Pedido*. O comprador e o pedido estão inicialmente ligados, mas o comprador e o pagamento não estão. Entretanto, após o método *getPagamento* ser executado, e uma instância de *Pagamento* ser retornada para o comprador, ele adquire visibilidade local para o pagamento, porque, neste ponto, o pagamento é atribuído a uma variável local (*umPagamento*) do método que o comprador está executando.

Deste momento em diante, a instância de *Comprador* adquire visibilidade local para o pagamento e pode se comunicar diretamente com ele.

Entretanto, um padrão de design conhecido como *Não fale com estranhos* (Larman, 2004)⁹ não recomenda que um objeto envie mensagens a objetos que são apenas localmente visíveis. De acordo com esse padrão, um comprador não deveria enviar mensagens para *umPagamento*. Como visto na Seção 9.6, existe sempre uma opção de design que evita esse tipo de comunicação. O problema aqui novamente é o acoplamento alto: se o comprador adquire visibilidade local para uma instância de *Pagamento*, ele adquire uma ligação que não é semanticamente suportada pelo modelo conceitual. No entanto, o preço a pagar pelo acoplamento baixo, nesse caso, é em alguns casos ter de implementar mais métodos, conforme mostrado adiante.

Um uso aceitável de visibilidade local ocorre quando um comando básico que cria um objeto é invocado. Quando um objeto é criado, a nova instância é usualmente referenciada por uma variável local. Esta visibilidade local pode ser imediatamente transformada em visibilidade por associação, como mostrado na Figura 9.32.

No exemplo, a primeira mensagem é um comando básico, um construtor que produz a instância de *Pagamento*. Naquele momento, a instância de *Pedido* obtém uma visibilidade local nova para a instância recém-criada, por meio da variável local *umPagamento*. A segunda mensagem liga aquela nova instância ao pedido, o qual adquire visibilidade por associação para ele.

A visibilidade local e por parâmetro são ambas válidas apenas dentro do escopo do método que as originou, assim como os parâmetros e variáveis locais em linguagens de programação são válidos apenas dentro dos métodos nos quais eles foram declarados, e

⁹ Uma simplificação da *Lei de Demétrio* (Leberherr; Holland, 1989).

desaparecem após os métodos terminarem seu escopo. Mas a visibilidade por associação é permanente, persistindo até que seja explicitamente removida por um comando básico que remove a ligação.

9.3.4 Visibilidade global

Existe *visibilidade global* para um objeto quando ele é acessível por qualquer outro objeto a qualquer tempo. O padrão de design *Singleton* (Gamma; Helm; Johnson; Vlissides, 1995) sugere que uma instância pode ser globalmente visível se ela for a única *instância de uma classe*. Isso faz sentido porque se esta classe tem uma única instância não é necessário criar associações de outros objetos para ela. Além disso, se aquela classe tem uma única instância, não é necessário passá-la como argumento para uma função porque a ideia com parâmetros é que eles podem variar, e isso não acontece com a classe que só pode ter uma única instância. Visibilidade global é a escolha nesse caso.

Exemplos de *singletons* são classes que representam serviços, tais como *Convertedor-DeMoedas*, *ContadorDeTempo*, *GeradorDeNúmeroDePedido* etc. Deve haver uma única instância da classe de serviço que possa ser acessada por qualquer objeto a qualquer tempo. Um exemplo é mostrado na Figura 9.33.

Se o padrão *Singleton* for abusado, ele pode se tornar um *antipadrão*. Designers não devem usar *singletons* para entidades do modelo conceitual; por exemplo, se o sistema for mono usuário e houver um único carrinho de compras de cada vez, ele não pode ser declarado como um *singleton*, porque ele é uma entidade conceitual e não uma classe de serviço. Uma entidade conceitual que é única hoje pode admitir múltiplas instâncias no futuro, e declará-la como *singleton* poderia prejudicar a evolução dessa classe.

Além disso, um *singleton*, em princípio, não deveria ter referências diretas para entidades¹⁰, porque isso reduziria sua reusabilidade potencial e criaria problemas de acoplamento alto. Classes de serviço não devem ser associadas com classes que são entidades; elas são globalmente visíveis, mas não devem disponibilizar acesso a nenhuma outra instância.

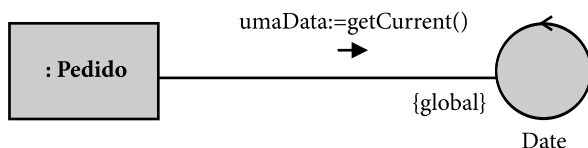


Figure 9.33

Visibilidade global.

¹⁰ Com exceção do controlador-fachada ou suas variações.

9.4 Modelagem dinâmica baseada em pós-condições

Vimos que contratos de comando de sistema apresentam um conjunto de pós-condições que correspondem a certos comandos básicos para criar e deletar instâncias, adicionar e remover ligações e modificar o valor de atributos. Esses contratos apenas indicam *o que* deve acontecer, mas eles não mostram *como* mensagens são trocadas entre objetos para obter essas metas. Os diagramas de interação UML podem ser usados para projetar exatamente como essas colaborações podem acontecer. Os seguintes princípios são recomendados:

- O tipo de visibilidade entre objetos depende fundamentalmente das características do papel, tais como multiplicidade, ordenação e qualificação, que são definidas no diagrama de classes e algumas vezes adicionalmente restringidas por precondições (Seção 9.3).
- O fluxo de controle em um diagrama dinâmico inicia na instância do controlador-fachada que recebe uma mensagem da interface.
- Quando um objeto que está executando não tem visibilidade para o objeto que deve realizar uma operação básica, ele deve *delegar* a responsabilidade e a execução a outro objeto que esteja mais perto daquele que pode realizar a responsabilidade.

Santos (2007) apresenta uma sistematização desses princípios ao definir um sistema de regras de produção de busca cega automática capaz de gerar diagramas de comunicação bem projetados a partir de uma grande seleção de contratos.

Em adição a esses princípios, padrões de design deveriam ser aplicados sempre que possível para ajudar o designer a construir métodos que sejam efetivamente reusáveis e manuteníveis.

9.4.1 Criação de instância

Quando um contrato estabelece que alguma instância foi criada, algum outro objeto deve efetivamente criá-la pelo envio de uma mensagem básica *Create* no respectivo diagrama de comunicação. O padrão de design *Criador* (Larman, 2004) estabelece que a escolha deveria ser preferencialmente:

- Uma instância de uma classe que tenha uma composição ou agregação compartilhada para a classe o objeto a ser criado.
- Uma instância de uma classe que tenha uma associação *um para muitos* para a classe do objeto a ser criado.
- Um objeto que tenha os dados de inicialização para o objeto a ser criado e que seja preferencialmente associado a ele.

Cada regra se aplica apenas se a anterior não se aplicar.

Para servir de referência para os exemplos seguintes, vamos retornar às operações do caso de uso 01: *Pedir livros*, apresentado no Capítulo 8. O modelo conceitual de referência da Figura 8.1 é repetido aqui, como mostra a Figura 9.34.

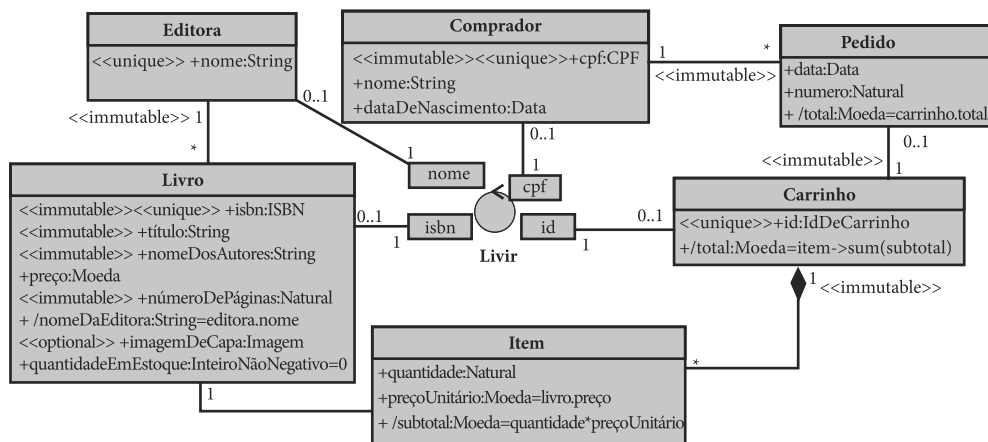

Figura 9.34

Diagrama de classes de referência.

O primeiro comando de sistema para o diagrama de sequência do caso de uso 01 é *adicionarAoCarrinho(umIdDeCarrinho:IdDeCarrinho, umISBN:ISBN, umaQuantidade:Natural)*. Seu contrato estabelece que, se o livro não está no carrinho ainda, um novo item é criado e ligado ao carrinho e ao livro. A *quantidade* do item é atualizada com o valor recebido como argumento, e *preçoUnitário* é atualizado como o preço do livro. Caso contrário, se o livro já está no pedido, uma nova quantidade é adicionada ao item corrente. O contrato, novamente, é o seguinte:

```

Context Livir::adicionarAoCarrinho(umIdDeCarrinho:IdDeCarrinho,
    umISBN:ISBN, umaQuantidade:Natural)
def: umCarrinho=carrinho[umIdDeCarrinho]
def: umLivro=livro[umISBN]
def: umItem=umCarrinho.item->select(livro.isbn=umISBN)
pre:
    umCarrinho->notEmpty
    umLivro->notEmpty
post:
    if umItem->isEmpty() then
        novoItem.isNewInstanceOf(Item) and
        umCarrinho^addItem(novoitem) and
        novoItem^setQuantidade(umaQuantidade) and
        novoItem^addLivro(umLivro) and
        novoItem^setPreçoUnitário(umLivro.preço)
    else
        umItem^setQuantidade(umItem.quantidade@pre+umaQuantidade)
    endif

```

Entre outras pós-condições, uma instância de *Item* deve ser criada se o livro ainda não estiver no carrinho. A questão neste momento é: se o fluxo de controle inicia no controlador-fachada (*Livir*), qual classe deve criar uma instância de *Item*? As opções seriam:

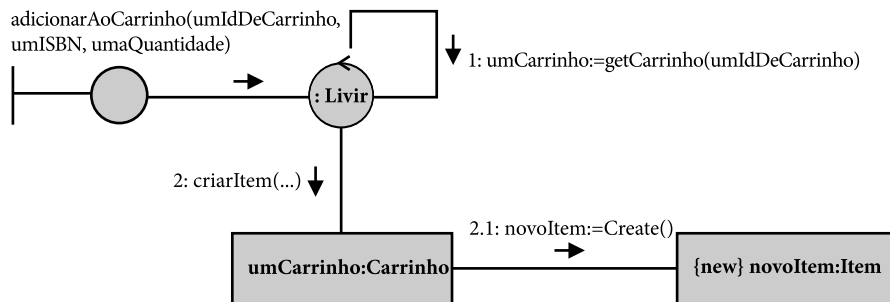
**Figura 9.35**

Diagrama de comunicação que mostra uma instância sendo criada.

- O próprio controlador-fachada, que já está com o fluxo de controle (essa poderia ser a escolha de designers inexperientes).
- Uma instância da classe *Livro*, porque ela tem uma associação de um para muitos para *Item*.
- Uma instância da classe *Carrinho*, porque ela é uma composição de *Item*.

Se a opção do controlador for escolhida, então um acoplamento que não existia antes entre ele e a classe *Item* será criado. Isso poderia ser uma má escolha de design.

De acordo com o padrão de design *Criador*, a melhor escolha seria a classe *Carrinho*. A associação entre *Livro* e *Item* é uma associação normal, enquanto a associação entre *Carrinho* e *Item* é uma composição.

Para a primeira versão do diagrama de comunicação, vamos esquecer por enquanto a condicional *if*, e as pós-condições que adicionam ligações e mudam o valor de atributos. Vamos nos concentrar em criar uma instância de *Item*.

A primeira tarefa para o designer é obter a instância corrente de *Carrinho* dado o seu *id* e delegar a ele a criação de uma nova instância de *Item*. Isso é mostrado na Figura 9.35.

Este diagrama implementa apenas uma pós-condição: a criação de uma instância de *Item*. Inicialmente, o controlador deve determinar qual é o carrinho identificado por *umIdDeCarrinho*. Isso é realizado pela mensagem rotulada com “1”, *getCarrinho*, a qual consulta o papel *carrinho* do controlador *Livir* e armazena a instância resultante na variável *umCarrinho*, a qual é uma variável local para o comando de sistema *adicionarNoCarrinho* que está sendo modelado. Como a pré-condição *umCarrinho->notEmpty()* assegura que *umIdDeCarrinho* é válido, o modelo dinâmico não precisa verificar isso novamente¹¹.

Na sequência, a mensagem rotulada “2”, *criarItem*, delega para o carrinho a responsabilidade de criação do item, porque o controlador não tem nenhuma associação com *Item*. Se o controlador tivesse criado o novo item sem delegar, então *Livir* necessitaria ser acoplado à classe *Item* e problemas relacionados ao alto acoplamento poderiam ocorrer.

¹¹ Isso é verdadeiro apenas em condições normais. Porém, se o sistema for sujeito a condições de concorrência complexas e se a segurança é um tópico crítico, algumas vezes a dupla verificação pode ser recomendável.

Os parâmetros de *criarItem* foram deixados para serem adicionados mais tarde já que eles ainda não são necessários.

Finalmente, a instância de *Carrinho* realiza a pós-condição por meio do envio da mensagem básica *Create*, rotulada com “2.1” no diagrama, a qual produz a instância de *Item*. Essa instância é armazenada em uma variável chamada *novolItem* que é local ao método *criarItem* na classe *Carrinho*.

Alguns designers poderiam preferir visualizar a colaboração entre objetos mostrada na Figura 9.35 como um diagrama de sequência. A Figura 9.36 apresenta o diagrama de sequência que é equivalente ao diagrama de comunicação da Figura 9.35.

Os diagramas de sequência e comunicação usados para modelagem dinâmica aqui têm três tipos de mensagens:

- Uma mensagem que invoca uma *operação de sistema*, a qual é enviada pela interface e recebida pelo controlador: *adicionarAoCarrinho* no exemplo.
- *Mensagens básicas* que invocam operações que não precisam ser mais refinadas: *getCarrinho* e *Create* no exemplo.
- *Mensagens delegadas* que podem ser usadas quando uma instância não tem visibilidade para o objeto que deve realizar a responsabilidade: *criarItem* no exemplo.

O primeiro e segundo tipos de mensagens estão sempre presentes nesses diagramas. A mensagem que invoca a operação de sistema é a raiz da árvore de mensagens do modelo dinâmico. As mensagens básicas são as folhas da árvore de mensagens, as quais correspondem às consultas e atualizações finais nos objetos.

Existe uma mensagem básica para cada pós-condição individual. Essas mensagens são *básicas* no sentido de que elas não precisam mais ser refinadas: o fluxo de controle cessa quando elas são atingidas.

As mensagens do terceiro tipo mencionado acima, *mensagens delegadas*, são opcionais. Elas podem ser usadas se o controlador, em vez de enviar as mensagens básicas ele próprio, deve delegar responsabilidades a outros objetos que têm visibilidade mais imediata para os objetos que devem ser consultados ou atualizados.

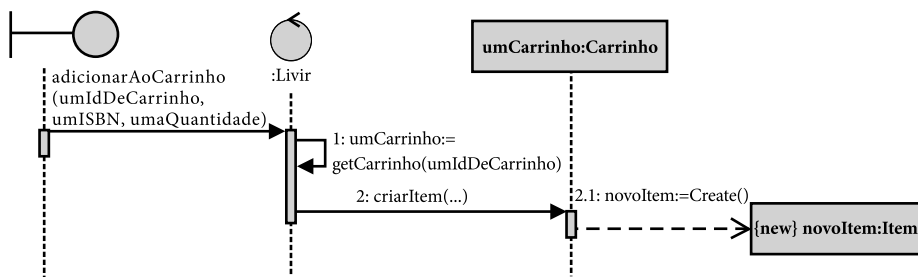


Figura 9.36

Diagrama de sequência equivalente ao diagrama de comunicação da Figura 9.35.

Mensagens delegadas são os ramos da árvore de mensagens, e o fluxo usualmente deve continuar depois delas. O fluxo de mensagens só termina quando uma mensagem básica é encontrada.

9.4.2 Adição de ligação

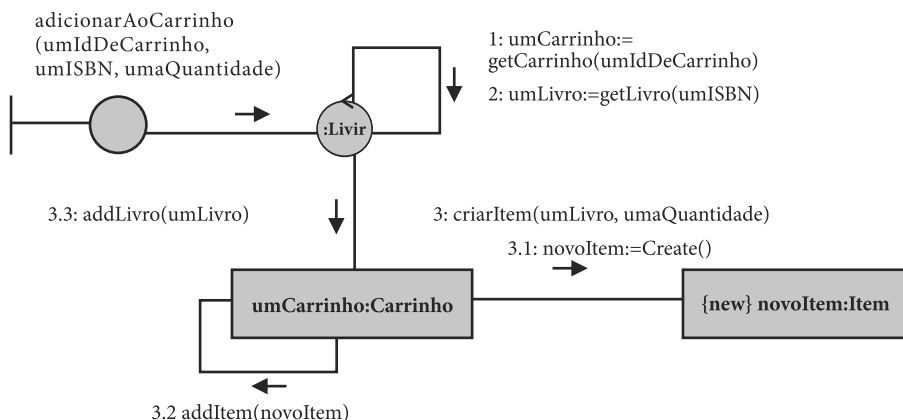
Foi mencionado que, quando uma instância é criada, ela deve ser imediatamente ligada a outra instância de forma que fique acessível a partir do controlador-fachada. O contrato para *adicionarAoCarrinho* apresentado na Seção 9.4.1 estabelece que as instâncias recém-criadas de *Item* devem ser ligadas a um livro e a um pedido. A Figura 9.37 mostra uma evolução do modelo dinâmico com essas duas pós-condições sendo obtidas.

Agora, quando o carrinho estiver executando o método *criarItem*, ele cria um novo item (mensagem 3.1), adiciona uma ligação entre ele próprio e o novo item (mensagem 3.2) e adiciona uma ligação entre o novo item e o livro identificado por *umISBN* (mensagem 3.3).

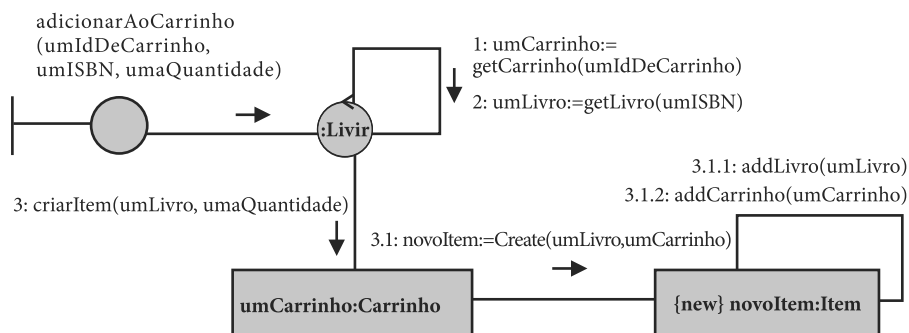
Este design poderia ser aceitável porque ele reproduz quase literalmente o que está especificado pelo contrato. Entretanto, esse só seria o caso se o código gerado por este design não fosse manualmente modificado posteriormente. Se o código de programação final vai sofrer manutenção direta em algum momento futuro, então precauções adicionais devem ser tomadas, e este é quase sempre o caso.

O problema é que quando um *Item* é criado por um construtor básico sem parâmetros, ele não está consistente com sua definição até que todo atributo e ligação obrigatórios sejam definidos para ele. No caso da Figura 9.37, *novoItem* é inconsistente no intervalo entre as mensagens 3.1 e 3.3. A solução para este problema é evitar construtores que retornam objetos inconsistentes. O construtor, nesse caso, deveria receber todos os parâmetros que são necessários para gerar uma instância consistente. Portanto, o construtor *Create* não é mais simplesmente uma mensagem básica na concepção inicial do termo, porque ele precisa ser refinado. A Figura 9.38 mostra uma nova versão do modelo dinâmico, na qual o construtor de *Item* produz uma instância com as associações obrigatórias necessárias.

Agora, mesmo o código de programação final pode ser mantido manualmente no futuro, o designer pode saber que a qualquer momento que uma instância de *Item* for usada ela estará consistente com sua definição.


Figura 9.37

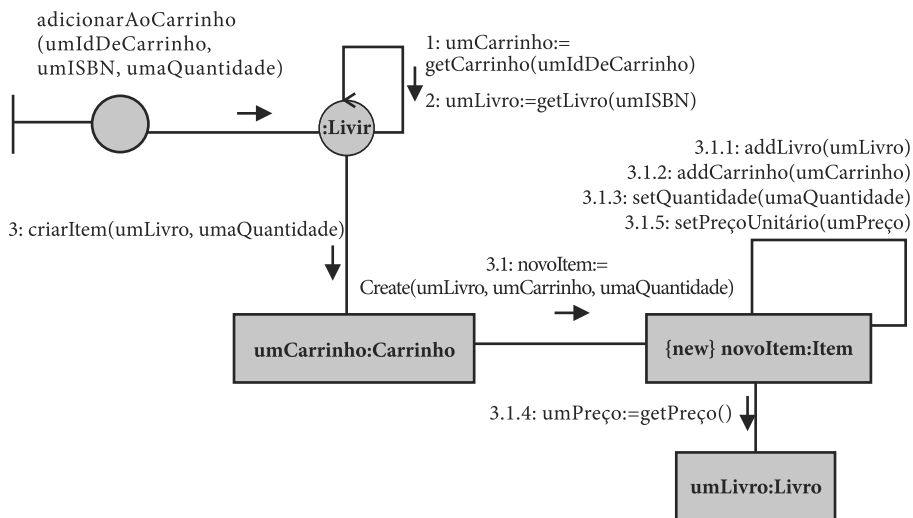
Evolução do modelo da Figura 9.35 mostrando a adição de ligações.


Figura 9.38

Uma variação do modelo da Figura 9.37 na qual o construtor de *Item* produz uma instância consistente.

9.4.3 Modificação de valor de atributo

Outro tipo de pós-condição que precisa ser considerado na modelagem dinâmica é a *modificação de valor de atributo*, a qual pode ser obtida pela mensagem básica *setter* que pode ser enviada pelo objeto que possui o atributo ou por um de seus vizinhos. Continuando o exemplo, o atributo *quantidade* para a nova instância de *Item* deve ser preenchido com o parâmetro *umaQuantidade* de *adicionarAoCarrinho*, e o atributo *preçoUnitário* de *Item* deve ser inicializado com o respectivo preço do livro, como estabelecido pela declaração do valor inicial para aquele atributo no diagrama. Essas pós-condições são adicionadas ao diagrama e mostradas na Figura 9.39.

**Figura 9.39**

Evolução do modelo na Figura 9.38 mostrando atributos sendo atualizados.

9.4.4 Destruição de instância

A destruição de uma instância é dinamicamente modelada pelo envio de uma mensagem básica *destroy* para um objeto. Os mesmos princípios do padrão de design *Criador* se aplicam aqui: o objeto que destrói uma instância deve ter uma agregação ou composição com o objeto que vai ser destruído, ou uma associação de um para muitos ou, pelo menos, estar ligado ao objeto.

Deve-se tomar cuidado para evitar deixar objetos com ligações obrigatórias inconsistentes após um objeto ser destruído. Se o contrato estiver bem formado, nenhum objeto órfão será deixado, e os diagramas de comunicação apenas necessitarão representar as pós-condições que foram já mencionadas no contrato.

A Figura 9.40 apresenta um modelo dinâmico para o comando CRUD *deletar* aplicado a uma instância de *Livro*. O contrato assume que apenas ISBNs de instâncias que podem ser deletadas são passadas como argumento¹²:

```

Context Livro::deletarLivro(umISBN:ISBN)
pre:
    livro[umISBN]->notEmpty() and
    livro[umISBN].item->isEmpty()
post:
    livro[umISBN]^destroy()
  
```

¹² Por exemplo, o comando *deletar* poderia ser chamado apenas para livros selecionados de uma lista que só contém livros que possam efetivamente ser deletados, isto é, livros sem itens ligados a eles.

9.4.5 Remoção e substituição de ligação

Quando uma pós-condição estabelece que uma ligação deve ser removida, algum objeto no modelo dinâmico deve enviar uma mensagem básica *remove* para uma das suas instâncias participantes da ligação. A Figura 9.42 mostra um exemplo de um modelo dinâmico baseado no diagrama de classes da Figura 9.41, no qual uma ligação é removida e substituída por outra; esta é a versão dinâmica do contrato que especifica como um carro troca de dono:

```
Context Controlador::substituirDono(cpfDoNovoDono:CPF,
umaPlaca:Placa)
pre:
    pessoa[cpfDoNovoDono]->notEmpty() and
    carro[umaPlaca]->notEmpty
post:
    carro[umaPlaca]^removeDono() and
    carro[umaPlaca]^addDono(pessoa[cpfDoNovoDono])
```

Como o papel *dono* tem multiplicidade 1, o comando que remove ou substitui a ligação não requer a especificação de um parâmetro.

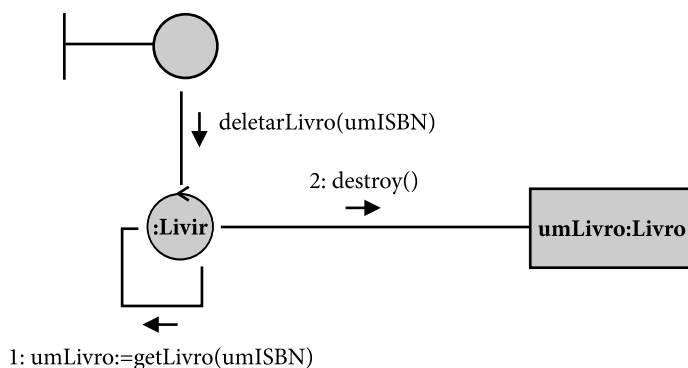


Figura 9.40

Diagrama de comunicação com um objeto sendo destruído.

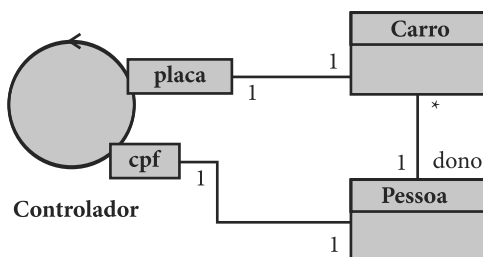


Figura 9.41

Diagrama de classes de referência.

Além disso, como o papel tem multiplicidade 1, uma nova mensagem básica para substituir a ligação pode ser criada em vez da sequência *remove/add*. Essa mensagem pode ser chamada de *setDono(novoDono:Pessoa)* ou *replaceDono(novoDono:Pessoa)*, e seria *thread-safe* (ou seja, mais segura em relação a processos concorrentes). A sequência *remove/add* poderia ser perigosa se no curto espaço de tempo entre esses dois comandos algum outro processo (*thread*) fizesse uma consulta no objeto que está temporariamente em estado inconsistente. Se um único comando como *replace* for usado em vez disso e se for garantido que enquanto ele estiver sendo realizado o objeto estará bloqueado mesmo para leitura, então essa situação perigosa descrita acima não poderá ocorrer.

9.4.6 Pós-condições condicionais

Como explicado anteriormente, algumas vezes, uma pós-condição deve ser obtida apenas se dadas condições forem válidas. Nesses casos, é possível usar mensagens condicionadas em diagramas de interação que são similares às condições de guarda usadas nos diagramas de máquina de estados e atividade.

Retomando o exemplo *adicionarAoCarrinho*, a expressão condicional verifica se o carrinho já tem um item com o livro dado. Essa expressão não se qualifica como um atributo derivado, porque ela é parametrizada (ela é booleana, mas seu valor depende do livro passado como parâmetro) e, portanto, ela pode ser definida como uma consulta normal na classe *Carrinho*:

```
Context Carrinho::incluiLivro(umLivro:Livro):Boolean
body:
    item.livro->exists(umLivro)
```

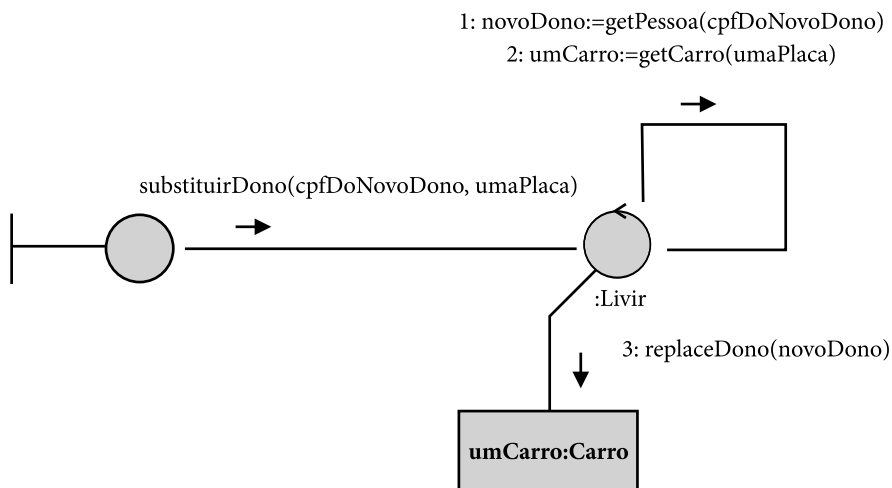
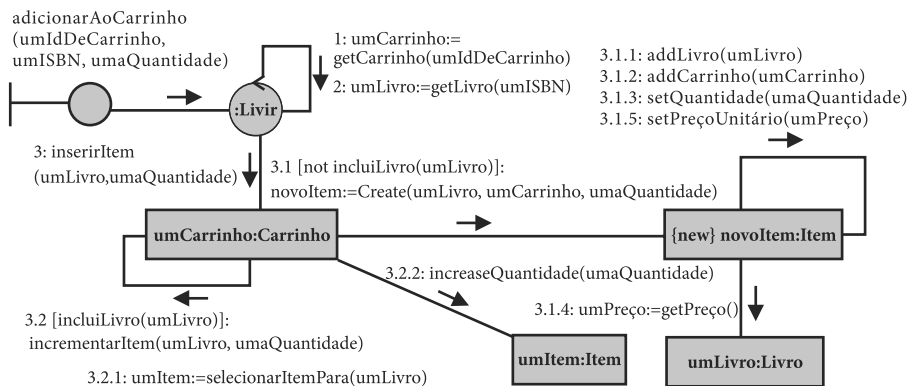


Figura 9.42

Diagrama de comunicação com um comando de substituição de ligação.


Figura 9.43

Um rascunho completo para o modelo dinâmico do contrato de *adicionarAoCarrinho*.

A Figura 9.43 mostra a evolução do modelo da Figura 9.39, agora testando se um livro já pertence ao carrinho ou não. Note que a mensagem 3 foi renomeada como *inserirItem* porque ela não é mais simplesmente a criação de um novo item: ele pode ser criado ou incrementado.

Se a condição fosse simplesmente um “*implies*”, então uma mensagem condicional poderia ser suficiente no diagrama, mas como a condição na Figura 9.43 envolve uma estrutura do tipo *if-then-else-endif*, duas mensagens condicionais mutuamente exclusivas devem ser usadas. Na Figura 9.43, a mensagem 3.1 é executada apenas se *incluiLivro(umLivro)* for falso, e a mensagem 3.2 é executada apenas se esta condição for verdadeira.

Note que as mensagens 3.1.1 a 3.1.5 são subordinadas a 3.1 e são executadas apenas se 3.1 o for. Similarmente, as mensagens 3.2.1 e 3.2.2 são subordinadas a 3.2 e são executadas apenas se 3.2 for executada.

A mensagem 3.2.1, *selecionarItemPara*, é outra consulta que pode ser útil para a classe *Carrinho*. Ela pode ser definida como:

```

Context Carrinho::selecionarItemPara(umLivro:Livro):Item
body:
    item->select(livro=umLivro)

```

A mensagem 3.2.2 é também importante: se apenas mensagens básicas (como definido anteriormente) forem usadas, em vez da mensagem 3.2.2, uma sequência *quantidadeAtual:=getQuantidade()* e *setQuantidade(quantidadeAtual+umaQuantidade)* seria usada. Entretanto, essa construção não é *thread-safe*. A menos que a concorrência seja perfeitamente controlada, essa sequência de mensagens poderia deixar a quantidade em um estado inconsistente se ela fosse atualizada entre duas mensagens por algum outro processo. Assim, usualmente, em vez de realizar uma sequência de *get* e *set* para incrementar atributos numéricos, é preferível definir uma mensagem básica *increase* que simplesmente adiciona a quantidade passada como parâmetro para o respectivo atributo numérico.

Até este ponto, a modelagem dinâmica produziu duas novas consultas para a classe *Carrinho*: *incluiLivro* e *selecionarItemPara*. Ela também produziu dois métodos delegados

para a classe *Carrinho*: *inserirItem* e *incrementarItem*. Para a classe *Item*, um construtor foi definido e a *quantidade* do item deve ser incrementada, uma nova mensagem básica também foi definida: *increaseQuantidade*. As outras mensagens no diagrama tais como *getCarrinho*, *setQuantidade* etc. são todas mensagens básicas.

Entretanto, outra preocupação deve ser tratada agora: os itens, após sua criação, podem ser atualizados? Esta questão está relacionada com o padrão de design *Imutabilidade* que estabelece que um objeto que não pode ser atualizado não deve implementar métodos de atualização públicos. Especificamente, no caso da classe *Item*, que tem uma instância sendo criada na Figura 9.43, devemos considerar se seus livros, carrinho, preço ou quantidade podem mudar depois. Isso depende, claramente, de outros modelos dinâmicos que serão construídos para outras operações de sistema, mas nós podemos assumir agora que *quantidade* e *preçoUnitário* podem mudar enquanto o livro e o carrinho ligados são imutáveis. Portanto, o item não pode manter *addLivro* e *addCarrinho* como métodos públicos. Métodos privados podem ser declarados em um diagrama de classes UML pela adição de um sinal de menos ("-") antes do nome do método. Eles são mostrados no diagrama da Figura 9.50. Métodos privados existem, mas somente podem ser chamados pela própria instância que os implementa; para outros objetos eles não são acessíveis.

9.4.7 Exceções

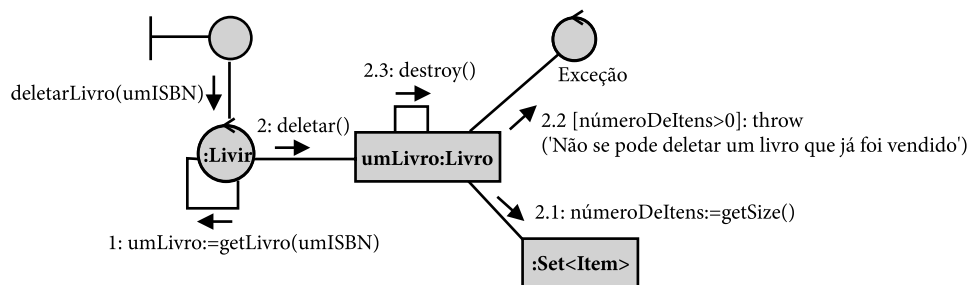
Condições também podem ser usadas para modelar exceções. Considere novamente o contrato para deletar um livro, mas dessa vez em vez de garantir por pré-condição que o livro pode ser deletado, uma exceção vai ocorrer se o livro estiver ligado a pelo menos um item:

```
Context Livro::deletarLivro(umISBN:ISBN)
pre:
    livro[umISBN]->notEmpty()
post:
    livro[umISBN]^destroy()
exception:
    livro[umISBN].item->notEmpty() implies
        Exception.throw('Não se pode deletar um livro que já foi
        vendido')13
```

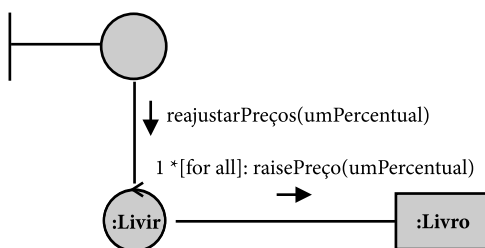
Exceções usualmente são as primeiras condições a serem testadas no modelo dinâmico, porque se uma delas ocorrer nenhuma das outras pós-condições pode ser obtida. Assim, uma mensagem condicional poderia ser usada para testar a condição da exceção antes de qualquer outra coisa.

O diagrama da Figura 9.44 mostra como testar e ativar uma exceção se o livro possuir itens.

¹³ Usualmente exceções não receberiam a mensagem de erro como parâmetro, mas um código de exceção que se referiria a uma lista de mensagens. Entretanto, para manter os exemplos mais simples de ler, elas são mantidas aqui do jeito que estão.


Figura 9.44

Um modelo dinâmico para um contrato com uma exceção.


Figura 9.45

Modelo dinâmico para um contrato com uma pós-condição sobre uma coleção de objetos.

O modelo da Figura 9.44 assume que a mensagem *throw* é terminal, isto é, a execução da operação de sistema é interrompida por ela. Isso significa que se a condição da exceção for verdadeira, a mensagem 2.3 (o verdadeiro *destructor*) não é executada.

9.4.8 Pós-condições sobre coleções

Quando comandos de sistema têm contratos com pós-condições que especificam atualizações sobre coleções de objetos, a estrutura de iteração pode ser usada no diagrama de comunicação para indicar que uma mensagem é iterativamente enviada a todos os elementos da coleção. Por exemplo, o seguinte comando de sistema aumenta o preço de todos os livros em *umPercentual*:

```

Context Livir::reajustarPreços(umPercentual:Percentual)
post:
    livro->forAll(umLivro|
        umLivro^setPreço(umLivro.preço@pre*(1+umPercentual))
    )

```

O diagrama da Figura 9.45 é um modelo dinâmico para este contrato.

Novamente, a escolha em definir uma nova mensagem básica *raisePreço* em vez da sequência de *getPreço* e *setPreço* deve-se ao fato de aquela opção ser *thread-safe*. A mensagem básica *raisePreço* atualiza o valor do atributo *preço* ao multiplicá-lo por 1 mais o parâmetro *umPercentual*.

Outra situação comum ocorre quando a iteração não deve ocorrer sobre todos os objetos na coleção, mas apenas sobre aqueles que satisfazem uma dada condição. O seguinte comando eleva em *umPercentual* os preços de livros com número de páginas maior do que o parâmetro *umNúmeroDePáginas*:

```
Context Livir: reajustarLivrosMaisGrossosQue(
    umPercentual: Percentual, umNúmeroDePáginas: Natural)
post:
    livro->select (númeroDePáginas>umNúmeroDePáginas)
    ->forall (umLivro|
        umLivro^setPreço (umLivro.preço@pre* (1+umPercentual))
    )
```

O diagrama de comunicação para este contrato é mostrado na Figura 9.46, a seguir.

9.5 Consultas de sistema

Os contratos de consulta de sistema usualmente definem apenas combinações de dados que são obtidas a partir de objetos. É possível, embora nem sempre útil, desenvolver diagramas de comunicação ou sequência para esses contratos, como, por exemplo, o diagrama de comunicação da Figura 9.47, que foi construído para o contrato da consulta de sistema *listarLivros* definida a seguir:

```
Context Livir::listarLivros(): Set
body:
    livro->collect (umLivro|
        Tuple {
            isbn=umLivro.isbn,
            título=umLivro.título,
            nomeDoAutor=umLivro.nomeDoAutor,
            preço=umLivro.preço,
            númeroDePáginas=umLivro.númeroDePáginas,
            nomeDaEditora=umLivro.editora.nome,
            imagemDeCapa=umLivro.imagemDeCapa,
            quantidadeEmEstoque=umLivro.quantidadeEmEstoque
        }
    )
```

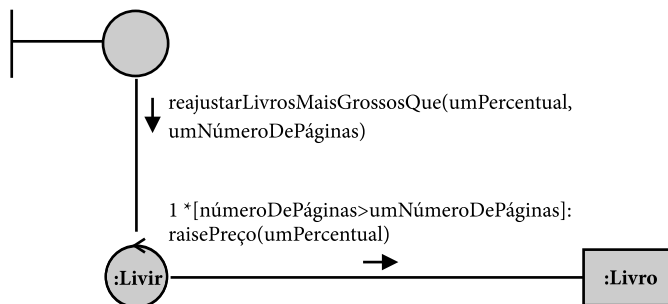


Figura 9.46

Diagrama de comunicação com iteração e seleção.

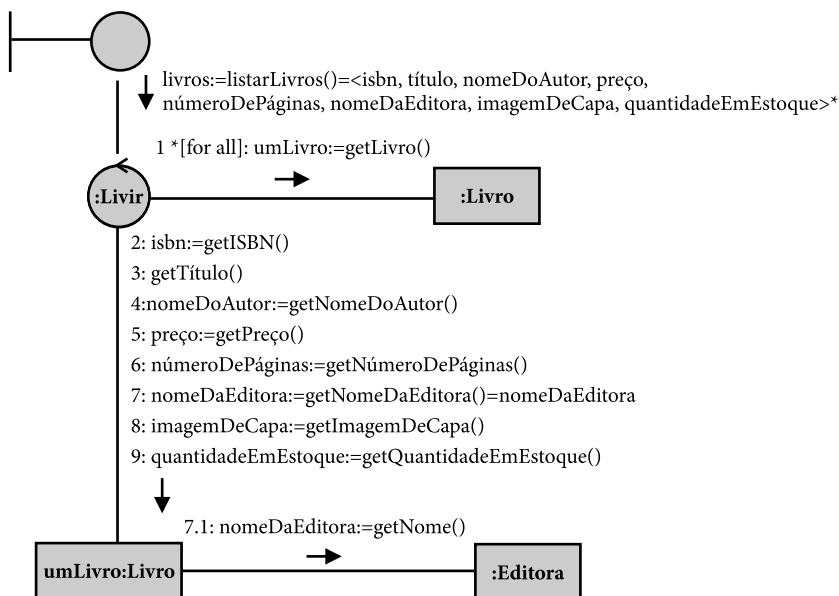

Figura 9.47

Diagrama de colaboração para uma consulta de sistema.

O aspecto mais difícil relacionado a esses diagramas quando eles são usados para modelar consultas de sistema é que eles não facilitam a representação de composição de dados. Por exemplo, na Figura 9.47 o controlador envia oito mensagens para cada instância de *Livro*; cada mensagem produz um retorno diferente. O que o controlador faz com todos esses valores? Sabemos, pelo contrato da consulta, que ele forma uma tupla com os oito valores e agrupa as tuplas em um conjunto que consiste no conjunto de dados de retorno para a consulta. Mas representar isso no diagrama apenas repetiria o que já se sabe a partir do contrato. Portanto, a utilidade destes diagramas no caso de consultas é limitada.

Existem algumas questões importantes sobre modelagem de consultas, porém. Primeiramente, três tipos de mensagens podem ser identificadas:

- *A consulta de sistema*, a qual é implementada no controlador e é chamada diretamente pela interface. Como os objetos de domínio são encapsulados dentro da camada de domínio e a interface vai lidar com campos e botões, consultas de sistema deveriam receber apenas dados alfanuméricos como parâmetros e retornar também apenas alfanuméricos.
- *Consultas básicas*, as quais consultam os valores reais de atributos e ligações correspondendo às consultas *get* ou *getters*.
- *Consultas intermediárias*, as quais muitas vezes correspondem aos atributos e associações derivadas. Atributos derivados retornam dados alfanuméricos e associações derivadas retornam objetos ou coleções que contêm objetos. Outro tipo de consulta intermediária é a consulta parametrizada, a qual não pode ser definida nem como

atributo derivado nem como associação derivada, como, por exemplo, *incluiLivros* na Figura 9.43.

Na Figura 9.47, a única consulta intermediária é *getNomeDaEditora*, enviada pelo controlador ao livro. O livro, por sua vez, envia a consulta básica *getNome* à sua editora. Assim, *nomeDaEditora* pode ser considerado um atributo derivado de *Livro*.

Com relação à questão, “A consulta intermediária deveria retornar o nome da editora ou a própria editora?”, algumas considerações devem ser feitas:

- Se um simples atributo de um objeto é necessário, evite retornar o objeto inteiro, porque isso pode gerar um risco de não conformidade com o padrão *Não fale com estranhos*. Se um único alfanumérico é necessário, então é melhor defini-lo como atributo derivado tal como *nomeDaEditora* na classe *Livro*.
- Quando um conjunto de atributos ou valores deve ser obtido a partir de um ou mais objetos, e a intenção é realizar uma operação sobre eles, por exemplo, realizando uma soma, ou encontrando o maior valor ou a média, é preferível realizar a operação na classe que tenha o acesso mais imediato aos valores necessários. Em outras palavras, a operação deveria ser realizada o mais cedo possível, e o resultado já processado deveria ser retornado como um atributo derivado. Por exemplo, *valorTotal*, que é um atributo derivado de *Carrinho*, deve ser calculado pelo próprio carrinho, e não pelo comprador ou pelo controlador. Seria um design bem ruim se o carrinho retornasse os itens para o controlador realizar a soma dos subtotais.
- Quando coleções de valores com uma estrutura complexa devem ser retornadas e não podem ser transformadas em um simples valor alfanumérico, é preferível retornar essa informação como uma coleção de objetos, definindo-a como uma associação derivada, porque dessa forma, os métodos que as classes já possuem para realizar os cálculos e operações sobre os dados permanecem disponíveis. Entretanto, esta escolha pode possivelmente criar mais acoplamento, e isso só deve ser usado se outras possibilidades realmente não são aplicáveis.

9.6 Delegação e acoplamento baixo

Até este ponto vimos algumas técnicas para construir diagramas de comunicação. Mas na maioria das situações há mais do que uma opção de design, e mais do que uma forma de enviar uma mensagem.

Como discutido anteriormente, algumas vezes, o objeto que possui o controle não tem visibilidade para o objeto que pode realizar a responsabilidade. Nesse caso, duas abordagens de design opostas podem ser identificadas:

- O objeto que tem o controle tenta obter uma referência ao objeto que pode realizar a responsabilidade, e comunica-se diretamente com ele.
- O objeto delega a mensagem a outro objeto que estiver mais próximo do objeto que pode realizar a responsabilidade.

Para entender a diferença entre essas duas abordagens, imagine que Maria é a chefe de Pedro. A chefe quer contratar um bufê para a festa do escritório, mas ela não conhece nenhuma empresa do ramo. A chefe sabe que Pedro tem contato com algumas empresas de bufê e, então, duas abordagens são possíveis:

- A chefe pede a Pedro os números de telefone das empresas e ela mesma faz os arranjos. A única coisa que Pedro fez foi passar seus contatos para a chefe.
- A chefe dá a Pedro os parâmetros para a festa: quantas pessoas, que tipo de comida, quanto gastar etc. e pede a Pedro que faça os arranjos. A festa acontece e a chefe nunca ficou sabendo qual o número de contato com a empresa de bufê: ela delegou a tarefa para Pedro.

Embora na vida real possa ser interessante manter contato com muitas pessoas e empresas, este não é o caso em sistemas orientados a objetos. Quando objetos estão altamente acoplados, o sistema fica mais complexo do que ele realmente precisa ser. Assim, é necessário evitar criar conexões entre objetos que não estivessem conectados anteriormente.

A primeira abordagem discutida anteriormente é chamada de *concentração*. Com ela, o objeto que está com o controle de execução procura obter referências para todos os objetos necessários e realiza a operação ele próprio. A segunda abordagem é chamada de *delegação* e consiste em fazer os objetos delegarem responsabilidade que eles não podem realizar diretamente, sem tentar obter acesso a outros objetos. A segunda abordagem é usualmente preferível, porque ela permite maior potencial de reusabilidade de classes e métodos, já que as consultas intermediárias e métodos delegados que essa abordagem cria usualmente são reusáveis.

Um exemplo concreto disso é mostrado a seguir. Considere uma consulta de sistema *getTotalDoCarrinho* definida como segue:

```
Context Livir::getTotalDoCarrinho(umIdDeCarrinho:IdDeCarrinho) :  
    Moeda  
    body:  
        carrinho[umIdDeCarrinho].item->sum(quantidade*preçoUnitário)
```

O contrato de consulta de sistema anterior tem como objetivo obter o valor total de um dado carrinho (a definição do contrato intencionalmente não usa os atributos derivados definidos na Figura 9.34, assumindo que eles ainda não foram definidos). Esse valor deve ser calculado a partir do preço e da quantidade de cada um dos itens do pedido.

Com a abordagem concentradora, o processo inteiro seria implementado na classe *Livir*. Essa classe deveria obter todos os dados necessários para realizar os cálculos e retornar o resultado para a interface. No diagrama de comunicação concentrador, todas as mensagens são enviadas pela instância *Livir*. O pseudocódigo para esta consulta de sistema poderia ser descrito como:

```

CLASSE Livir
  ASSOCIAÇÃO VAR carrinhos:MAP[IdDeCarrinho->Carrinho]
  ...
  MÉTODO getTotalDoCarrinho():Moeda
    LOCAL VAR umCarrinho: Carrinho
    LOCAL VAR itensDoCarrinho:Set<Item>
    LOCAL VAR total:Moeda
    umCarrinho:=carrinhos.at(umIdDeCarrinho)
    itensDoCarrinho:=umCarrinho.getItem()
    total:=0
    PARA CADA item EM itensDoCarrinho FAÇA
      total:=total+(item.getValor()*item.getQuantidade())
    FIM PARA
  RETORNA total
FIM MÉTODO
FIM CLASSE

```

Pode-se concluir o seguinte a partir desse código:

- Ele resolve o problema.
- Ele não produz nenhum elemento de software reusável a não ser a própria consulta que é implementada na classe *Livir*.
- Ele aumenta o acoplamento entre as classes, porque *Livir* agora requer visibilidade para *Item*, o que não era o caso no diagrama de classes da Figura 9.34.

Usando-se a abordagem da delegação, a consulta pode ser implementada corretamente da mesma forma, mas ao mesmo tempo novas estruturas reusáveis com maior coesão e menos acoplamento são criadas.

Em primeiro lugar, a classe *Livir* precisa evitar ter acesso a qualquer instância de *Item*, já que não há conexões entre essas classes no diagrama de classes. Para realizar as responsabilidades necessárias, *Livir* deve delegar o cálculo para o carrinho. Como o valor resultante é *Moeda* (um alfanumérico específico), a criação de uma consulta intermediária para a classe *Carrinho* é equivalente a definir um atributo derivado para ela com a seguinte especificação OCL:

```

Context Carrinho::total:Moeda
  derive:
    item->sum(subtotal)

```

Além de criar um atributo derivado em *Carrinho*, um atributo derivado também pode ser criado na classe *Item*, para retornar o subtotal:

```

Context Item::subtotal:Moeda
  derive:
    preçoUnitário*quantidade

```

Agora, com esses dois atributos derivados, o pseudocódigo pode ser definido com responsabilidades distribuídas entre as classes:

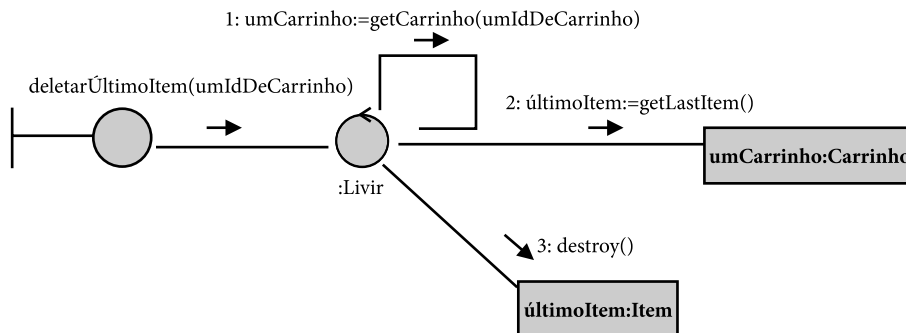
```

CLASSE Livir
  ASSOCIAÇÃO VAR carrinhos:MAP[IdDeCarrinho->Carrinho]
  ...
  MÉTODO getTotalDoCarrinho(umIdDeCarrinho:IdDeCarrinho):Moeda
    RETORNA carrinhos.at(umIdDeCarrinho).getTotal()
  FIM MÉTODO
FIM CLASSE
CLASSE Carrinho
  ASSOCIAÇÃO VAR itens:SET<Item>
  ...
  MÉTODO getTotal():Moeda
    LOCAL VAR total:Moeda
    total:=0
    PARA CADA item EM itens FAÇA
      total:=total+item.getSubtotal()
    FIM PARA
    RETORNA total
  FIM MÉTODO
FIM CLASSE
CLASSE Item
  ATRIBUTO VAR quantidade:Natural
  ATRIBUTO VAR preçoUnitário:Moeda
  ...
  MÉTODO getSubtotal():Moeda
    RETORNA quantidade*preçoUnitário
  FIM MÉTODO
FIM CLASSE

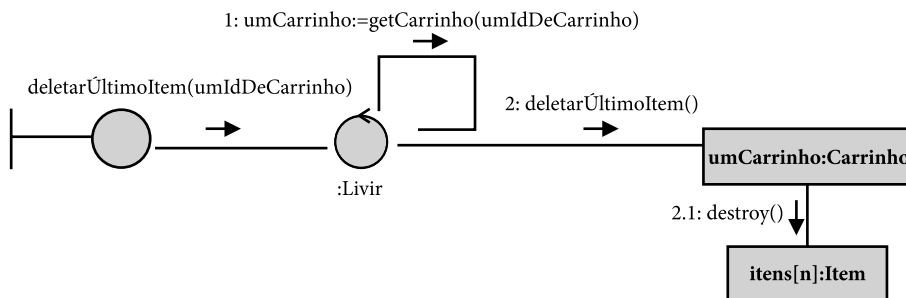
```

Com essa abordagem, a maior parte do cálculo acontece na classe que efetivamente tem acesso imediato à informação necessária, que é *Carrinho*. Assim, o procedimento de cálculo aqui é mais reusável do que aquele produzido pela abordagem concentradora, porque agora ele está dentro da classe *Carrinho*. Se a classe *Carrinho* for reusada, ela pode calcular seu próprio total sem depender de uma classe que não é sua componente (*Livir*). O mesmo é verdadeiro para *Item*, que agora sabe como calcular seu próprio subtotal.

No caso de comandos de sistema, o princípio de acoplamento baixo também é obtido quando o designer escolhe delegar em vez de retornar um objeto. Considere novamente o diagrama de classes da Figura 9.34, mas suponha que o papel de *Carrinho* para *Item* seja marcado com {ordered}. O diagrama de comunicação da Figura 9.48 mostra o comando de sistema *removerÚltimoItem*, usando a abordagem concentradora. A Figura 9.49 mostra o mesmo comando com a abordagem de delegação, evitando acoplamentos desnecessários entre as classes.

**Figura 9.48**

Estilo de design concentrador.

**Figura 9.49**

Estilo de design com delegação.

O preço de usar delegação é usualmente ter mais métodos declarados nas classes intermediárias. Entretanto, métodos reusáveis são usualmente produzidos para compensar essa questão. O preço de usar a abordagem concentradora é ter um design no qual responsabilidades estão concentradas no controlador-fachada em vez de estarem distribuídas entre os objetos; isso produz um modelo que pode ser difícil de manter e evoluir a longo prazo.

9.7 Diagrama de classes de design

Um dos objetivos do design lógico é construir e refinar o diagrama de classes de design (DCD), o qual é criado a partir do modelo conceitual e da informação obtida durante a modelagem dinâmica (quando os diagramas de interação para os contratos estão sendo construídos).

A primeira versão do DCD é uma cópia exata do modelo conceitual, a qual pode ser modificada mais tarde. As modificações básicas que são feitas no DCD durante o design da camada de domínio são as seguintes:

- *Métodos são adicionados.* Durante a análise, apenas comandos e consultas de sistema foram descobertos e adicionados à classe do controlador-fachada. Durante o design, métodos básicos e delegados pertencentes a outras classes são adicionados.
- *Associações são direcionadas.* Durante a análise, associações eram não direcionais. Durante o design sua direção é determinada pela direção das mensagens fluindo entre objetos nos diagramas de interação.
- *Atributos e associações podem ser detalhados.* Não é obrigatório apresentar detalhes sobre atributos e associações durante a análise. Esses detalhes podem ser adicionados durante o design. Adicionalmente, na análise apenas tipos abstratos de dados são usados para especificar papéis de associação, e eles podem ser substituídos por tipos de dados concretos durante o design (por exemplo, trocando *sequence* por *array* ou *linked list*).
- *A estrutura das classes e associações pode mudar.* Pode ser necessário criar novas classes para implementar certas estruturas de design, tais como estratégias e serviços, por exemplo. Assim, é possível que a estrutura de classes do DCD não corresponda exatamente à estrutura do modelo conceitual em alguns casos.
- *Possibilidade de criação de atributos privados e protegidos.* No modelo conceitual, todos os atributos são públicos, porque eles representam informação estática e não comportamento. Entretanto, quando os aspectos dinâmicos dos objetos são representados, pode ser necessário trabalhar com atributos privados para encapsular estados internos que determinam o comportamento de alguns objetos. Esses atributos podem algumas vezes não ser diretamente acessíveis por meio de um *getter* ou *setter*, mas sua influência nos métodos é sentida.

Durante a construção de um diagrama de comunicação, cada vez que um objeto recebe uma mensagem, sua classe deve implementar o método correspondente.

Na Figura 9.43, o diagrama de comunicação mostra que um conjunto de mensagens deve ser implementado como métodos em algumas classes:

- *Livro* deve implementar:
 - *adicionarAoCarrinho(umIdDeCarrinho:IdDeCarrinho, umISBN:ISBN, umaQuantidade:Natural)* – o comando de sistema.
 - *getCarrinho(umIdDeCarrinho:IdDeCarrinho):Carrinho* – uma consulta básica.
 - *getLivro(umISBN:ISBN):Livro* – uma consulta básica.
- *Carrinho* deve implementar:
 - *inserirItem(umLivro:Livro, umaQuantidade:Natural)* – comando delegado.
 - *incluiLivro(umLivro:Livro):Booleano* – consulta ou predicado.

- *incrementarItem(umLivro:Livro, umaQuantidade:Natural)* – comando delegado.
- *selecionarItemPara(umLivro:Livro):Item* – consulta.
- *Item* deve implementar:
 - *Create(umLivro:Livro, umCarrinho:Carrinho, umaQuantidade:Natural)* – construtor.
 - *addLivro(umLivro:Livro)* – comando básico privado.
 - *addCarrinho(umCarrinho:Carrinho)* – comando básico privado.
 - *setQuantidade(umaQuantidade:Natural)* – comando básico.
 - *increaseQuantidade(umaQuantidade:Natural)* – comando básico.
 - *setPreçoUnitário(umValor:Moeda)* – comando básico.
- *Livro* deve implementar:
 - *getPreço():Moeda* – consulta básica.

O DCD pode também incluir informação sobre a direção das associações se o designer escolher não implementar todas as associações como bidirecionais. Este é usualmente o caso porque associações unidirecionais tendem a ser mais simples de implementar e são mais eficientes quando comparadas com associações bidirecionais.

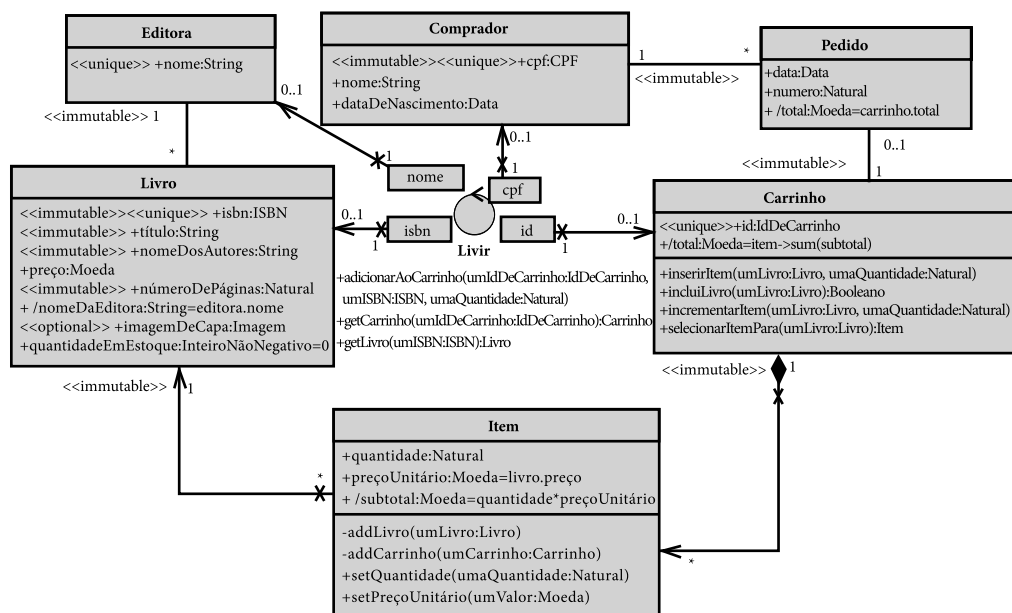


Figura 9.50

Diagrama de classes de design com métodos e direção de associações.

Decidir se uma associação é unidirecional ou bidirecional depende do fluxo de mensagens dos diagramas de interação. Nenhuma associação deve ser, em princípio, navegável na direção do controlador: o controlador vê as classes, mas elas não o veem. Outras associações podem ser unidirecionais se as mensagens apenas fluem em uma direção, e bidirecionais se as mensagens fluem em diferentes direções, mesmo se isso ocorrer em diferentes diagramas. A Figura 9.50 apresenta uma evolução do diagrama da Figura 9.34 na qual métodos e direção de associações descobertos na modelagem dinâmica da Figura 9.39 foram incluídos.

Nem toda associação na Figura 9.50 é direcionada porque o diagrama da Figura 9.39 mostra mensagens sendo enviadas apenas do controlador para *Carrinho*, de *Carrinho* para *Item* e de *Item* para *Livro*. Associações que não foram rotuladas com mensagens no diagrama de comunicação são deixadas indefinidas até que outro diagrama eventualmente decida sobre suas direções.

Se outros diagramas de colaboração desenvolvidos para diferentes operações de sistema necessitarem que mensagens naveguem na direção oposta à que já foi descoberta, então a respectiva associação deverá ser bidirecional.

Como a direção das associações depende da direção das mensagens enviadas nos modelos dinâmicos, não faria sentido tentar definir essa direção durante a modelagem conceitual preliminar (Capítulo 3). Naquele ponto da análise, a informação disponível ainda é insuficiente para tomar essas decisões de design. O mesmo é válido para decidir quais são os métodos que uma classe deve implementar.

Assume-se por *default* que as associações unidirecionais não têm restrições em sua origem. Entretanto, este nem sempre é o caso. Isso é verdade, por exemplo, para a associação de *Item* para *Livro* que não tem restrição de multiplicidade na origem. Mas a associação de *Carrinho* para *Item* é restrita na origem porque ela requer que um item seja ligado a exatamente um carrinho. Se a associação fosse implementada simplesmente como uma variável na classe *Carrinho* contendo um conjunto de *Item*, então não seria muito fácil garantir esta restrição. É por isso que no próximo capítulo associações unidirecionais com restrições na origem são tratadas como associações bidirecionais: elas podem não ser navegáveis em ambas as direções, mas precisam ser implementadas nos dois lados.

As atividades de design lógico terminam quando o DCD tem informação suficiente para implementar as classes da camada de domínio. Isso usualmente acontece quando todos os contratos de operação de sistema tiverem sido examinados e seus modelos dinâmicos incorporados no design.

9.8 O processo visto aqui

	Concepção	Elaboração
Modelagem de negócio		
Requisitos		
Análise e Design		Fazer o design da camada de domínio: • Criar diagramas de sequência ou comunicação para cada contrato de comando de sistema. • Usar este diagrama para decidir quais métodos implementar em cada classe. • Inspeccionar os diagramas para descobrir quais associações podem ser unidirecionais e defini-las como tal. • Observar os contratos de consulta de sistema e decidir quais consultas delegadas deveriam ser implementadas; algumas delas poderiam ser atributos ou associações derivados. • Produzir ou refinar o diagrama de classes de design.
Implementação		
Teste		
Gerenciamento de Projeto		

9.9 Questões

1. Quais são os seis tipos de responsabilidade que os objetos podem ter? Qual tipo de método é usado para realizar cada responsabilidade?
2. Quais são os quatro tipos de visibilidade que podem existir entre objetos? Elas são simétricas? Como cada uma delas influencia o acoplamento alto entre as classes?
3. Leia novamente o último parágrafo da Seção 9.3.2. Tente desenhar um diagrama de comunicação que ilustre aquela situação. Por que este design seria pior do que o diagrama apresentado na Figura 9.29?
4. Explique a influência das precondições e exceções de contratos em modelos dinâmicos. Como elas afetam a complexidade do modelo dinâmico?
5. Desenhe um diagrama de sequência equivalente ao da Figura 9.43.

Geração de código

10

TÓPICOS-CHAVE NESTE CAPÍTULO

- Geração de código para classes, atributos e associações
- Geração de código para métodos delegados e operações de sistema
- Padrões para consultas com filtros

10.1 Introdução à geração de código

A disciplina de *Implementação* do UP inclui atividades de geração de código para modelos de design. É necessário gerar código para classes da camada de domínio e também para as outras camadas tecnológicas do sistema. Este capítulo concentra-se na geração de código para a camada de domínio.

Uma vez que os diagramas de comunicação e o DCD tenham sido produzidos, a geração de código é uma atividade que pode ser sistematizada a um ponto em que ela pode praticamente ser feita de forma automática. Geração automática de código é possível para classes de domínio que realizam todo o processamento lógico especificado pelos contratos de operação de sistema.

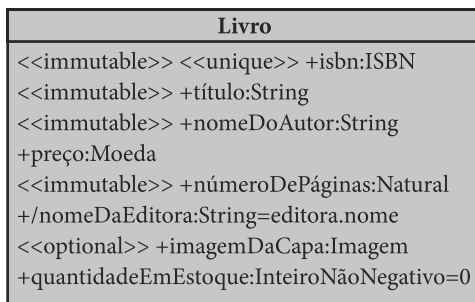
Este capítulo apresenta regras para geração de código para o DCD e diagramas de comunicação. Os exemplos são apresentados em pseudocódigo, o qual pode ser traduzido para a maioria das linguagens de programação (preferencialmente orientadas a objetos).

10.2 Classes e atributos

As classes do DCD são usualmente convertidas de forma direta em classes de linguagens de programação. Atributos das classes são convertidos em *variáveis de instância privadas* na respectiva classe.

Como explicado no Capítulo 6, os tipos dos atributos devem ser alfanuméricos (tais como *Integer*, *Real*, *String*, *Booleano* etc.), primitivos (tais como *Data*, *Moeda*, *ISBN* etc.) ou enumerações (tais como *DiaDaSemana*, *Gênero*, *TipoDeTelefone* etc.).

Se outros objetos podem acessar o atributo, então ele deve ser implementado com um método *getter*. Se o atributo puder ser atualizado, então ele deve ser implementado com um *setter*, o qual seria um método básico do tipo *setAtributo* ou uma variação como *incrementAtributo*, se o atributo for numérico, por exemplo. A Figura 10.1 apresenta um exemplo de uma classe de design que é usada para mostrar como o código de programação é gerado.

**Figura 10.1**

Classe de referência para geração de código.

O código seguinte corresponde ao pseudocódigo de implementação para a classe mostrada na Figura 10.1. Primeiramente, os atributos da classe são transformados em atributos privados no código de programação:

```

CLASSE Livro
  VAR ATRIBUTO PRIVADO isbn:ISBN
  VAR ATRIBUTO PRIVADO título:String
  VAR ATRIBUTO PRIVADO nomeDoAutor:String
  VAR ATRIBUTO PRIVADO preço:Moeda
  VAR ATRIBUTO PRIVADO númeroDePáginas:Natural
  VAR ATRIBUTO PRIVADO imagemDaCapa:Imagem
  VAR ATRIBUTO PRIVADO quantidadeEmEstoque:InteiroNãoNegativo
  ...

```

Observe que todos os atributos, exceto *nomeDaEditora*, o qual é privado, foram definidos como atributos no pseudocódigo. A maioria das linguagens não diferencia **VAR ATRIBUTO** de **VAR ASSOCIAÇÃO**. Neste livro, esses nomes são usados para deixar clara a distinção entre variáveis que representam atributos e variáveis que implementam papéis de associação. Entretanto, para a maioria das linguagens, qualquer variável de instância será declarada simplesmente com **VAR** ou alguma expressão equivalente específica da linguagem.

No entanto, a maioria das linguagens diferencia variáveis e métodos do tipo **PRIVADO** em oposição ao tipo **PÚBLICO**. Sempre que variáveis privadas são permitidas, elas são a melhor escolha, porque, dessa forma, atributos ficam encapsulados dentro do objeto e eles não poderão ser atualizados por outros objetos que não tenham autorização para isso.

Agora, devemos considerar como os atributos da classe vão ser atualizados. Alguns deles podem ser atualizados a qualquer momento, enquanto outros podem ser definidos apenas no instante da criação do objeto e sua atualização é vedada depois disso: esses são os *atributos imutáveis*. Na Figura 10.1, atributos que não podem mudar depois que o objeto é criado são estereotipados como `<<immutable>>`.

Um conjunto de *setters* e um¹ *construtor* podem ser definidos seguindo as seguintes recomendações:

- O construtor deve receber como parâmetros os valores iniciais de todos os atributos que não sejam opcionais, derivados ou com valor inicial definido (se houver).
- A implementação do construtor inicializa os atributos com valores iniciais usando os respectivos valores definidos (se houver).
- O construtor deve receber como parâmetros os objetos que devem preencher os papéis de associação obrigatórios (se houver) – mais tarde será mostrado um exemplo mais completo sobre isso.
- Apenas atributos que não são derivados ou imutáveis devem definir um ou mais *setters* (*setAtributo*, *incrementAtributo* etc., dependendo do caso).

Assim, o seguinte código poderia ser criado como uma continuação da definição da classe de implementação que iniciou anteriormente:

```
MÉTODO CONSTRUTOR Create(umISBN:ISBN,  
    umTítulo, umNomeDeAutor:String, umPreço:Moeda,  
    umNúmeroDePáginas:Natural)  
    isbn:=umISBN  
    título:=umTítulo  
    nomeDoAutor:=umNomeDeAutor  
    preço:=umPreço  
    númeroDePáginas:=umNúmeroDePáginas  
    quantidadeEmEstoque:=0  
FIM MÉTODO CONSTRUTOR  
MÉTODO setPreço(umPreço:Moeda)  
    preço:=umPreço  
FIM MÉTODO  
MÉTODO raisePreço(umPercentual:Percentual)  
    preço:=umPreço*(1+umPercentual)  
FIM MÉTODO  
MÉTODO setImagemDaCapa(umaImagemDeCapa:Imagem)  
    imagemDaCapa:=umaImagemDeCapa  
FIM MÉTODO  
MÉTODO increaseQuantidadeEmEstoque(umIncremento:Integer)  
    quantidadeEmEstoque:=quantidadeEmEstoque+umIncremento  
FIM MÉTODO  
...
```

Note que dentro do construtor cada atributo é atualizado para o respectivo argumento e que para *quantidadeEmEstoque* é atribuído 0 (seu valor inicial *default*).

Apenas três atributos podem ser atualizados. O atributo *preço* tem dois *setters*: *setPreço*, o qual define um novo preço independentemente do valor original, e *raisePreço*, o qual aumenta o preço baseado em uma porcentagem.

¹ Em alguns casos, classes podem exigir mais do que um construtor.

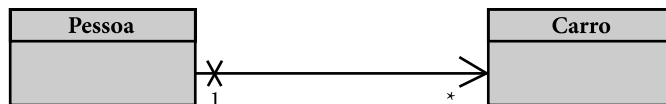
O atributo *quantidadeEmEstoque* é atualizado apenas pela adição de um incremento a ele. Como o parâmetro *umIncremento* pode ser negativo, decrementos podem ser feitos com este método também.

Vamos supor que agora o design produziu diagramas de interação que demonstram que todos os atributos dessa classe devem ser acessíveis por outros objetos. Nesse caso, todos os atributos devem implementar um método *getter*, mesmo os derivados. Assim, o código a seguir poderia ser uma continuação do código iniciado anteriormente:

```
...
MÉTODO getISBN():ISBN
    RETORNA isbn
FIM MÉTODO
MÉTODO getTítulo():String
    RETORNA título
FIM MÉTODO
MÉTODO getNomeDoAutor():String
    RETORNA nomeDoAutor
FIM MÉTODO
MÉTODO getPreço():Moeda
    RETORNA preço
FIM MÉTODO
MÉTODO getNúmeroDePáginas():Natural
    RETORNA título
FIM MÉTODO
MÉTODO getNomeDaEditora():String
    RETORNA ...2
FIM MÉTODO
MÉTODO getImagemDaCapa():Imagem
    RETORNA imagemDaCapa
FIM MÉTODO
MÉTODO quantidadeEmEstoque():InteiroNãoNegativo
    RETORNA quantidadeEmEstoque
FIM MÉTODO
```

Neste ponto, o leitor poderia estar se perguntando por que *nomeDoAutor* ainda é um atributo tipado como *String* e não uma classe com uma associação de “muitos para muitos” com *Livro*. A razão é que este exemplo está simulando um projeto em andamento, e não a apresentação da versão final dele. Até este ponto, os únicos casos de uso examinados não exigiram que autores fossem tratados como conceitos complexos. O atributo *nomeDoAutor* é perfeitamente adequado para as funcionalidades descobertas até este ponto. Poderíamos até acreditar que isso poderia mudar quando outros casos de uso forem examinados no futuro, mas como William de Ockham (1494) disse “*Numquam ponenda est pluralitas sine necessitate*” (“nunca se deve colocar a mais sem necessidade”). Isso também adere ao princípio ágil de nunca implementar funcionalidades antes que elas se tornem necessárias só porque seria fácil fazer isso agora.

² Este método envolve associações, que serão explicadas nas Seções 10.3 e 10.4.


Figura 10.2

Associação unidirecional com multiplicidade restrita no papel de origem.

10.3 Associações unidirecionais

Associações unidirecionais sem restrições de multiplicidade na origem e associações unidirecionais a partir de *singletons*, tais como o controlador-fachada, podem ser implementadas de forma similar a atributos como *variáveis de instância* e, se necessário, elas deveriam ter métodos para ser atualizadas e consultadas.

Entretanto, se existem restrições de multiplicidade no papel de origem, essas associações devem ser tratadas como *bidirecionais*, mesmo se elas forem navegáveis em apenas uma direção. A Figura 10.2 mostra um exemplo desta situação, no qual embora a associação seja apenas navegável de *Carro* para *Pessoa*, todo carro deve estar ligado a uma única pessoa. Se a associação for implementada como uma variável de instância na classe *Pessoa*, será difícil assegurar que um carro pertence a uma mesma pessoa.

Existem ainda algumas considerações que devem ser discutidas sobre as diferenças entre um atributo e uma associação unidirecional. Primeiramente, atributos são sempre implementados por variáveis, cujos tipos são alfanuméricos, primitivos ou enumerações. Associações, no entanto, são implementadas por variáveis, cujos tipos são classes de domínio (no caso de associações *para um*) ou estruturas de dados (no caso de associações *para muitos*).

Adicionalmente, considerando-se diferentes multiplicidades de papel e outras características de associações, há distinções a serem feitas em relação aos métodos a serem implementados para cada tipo de associação.

O código gerado para associações e atributos temporários ou persistentes é o mesmo. A única diferença entre esses tipos de elementos reside na forma como eles são armazenados.

Em geral e se necessário, uma classe pode implementar três tipos de métodos para cada associação:

- Métodos para adicionar ligações: usualmente referidos como *addPapel*.
- Métodos para remover ligações: usualmente referidos como *removePapel*.
- Métodos para consultar objetos ligados: usualmente *getPapel* poderia ser implementado para retornar o conjunto completo de objetos ligados. Este conjunto *deve ser protegido* no sentido de que ele pode ser consultado e iterações podem ser feitas sobre seus elementos, mas nenhum elemento pode ser adicionado ou removido do conjunto original. Adicionalmente, métodos para obter um elemento ou subconjunto específico dada uma chave ou outro critério de busca, podem ser implementados, se necessário.

Usualmente, associações para um podem ser implementadas como uma variável simples e não como uma coleção. Nesse caso, os métodos *addPapel* e *removePapel* não se aplicam e a classe implementa, em vez deles um método *replacePapel* que substitui a ligação corrente por uma nova ligação.

Associações para 0..1 também requerem a implementação do método *replacePapel*. Mas nesse caso, *addPapel* e *removePapel* podem ser implementadas também. Se os métodos *addPapel* e *removePapel* forem implementados para papéis 0..1, então elas devem verificar os limites da multiplicidade.

Associações derivadas devem implementar apenas o método *get*, de acordo com sua definição.

Outros métodos ainda poderiam ser necessários dependendo do tipo de associação, tais como:

- Se a associação é *qualificada*, pode existir um *get* adicional que recebe como argumento a chave do qualificador e retorna o objeto ou subconjunto correspondente. Adicionalmente, se o qualificador for externo, o método *add* deveria receber esse valor para definir o qualificador; se ele for um qualificador interno, este argumento não precisa ser passado porque ele pode ser obtido a partir do próprio objeto. Um novo método *remove* também poderia ser implementado para remover uma ligação para um objeto baseando-se no valor de seu qualificador.
- No caso de associações *ordenadas*, um método *get* adicional pode retornar um objeto baseado em sua posição na coleção. Além disso, o método *add* poderia adicionar elementos em uma dada posição, a qual seria indicada como um novo parâmetro para o método, e o método *remove* poderia remover uma ligação de uma dada posição. Associações ordenadas poderiam também ter métodos para acessar, adicionar e remover elementos de seu início e fim.
- *Pilhas* e *filas* podem ter métodos especiais, tais como *push* e *pop* que seguem as regras específicas dessas estruturas.

A Tabela 10.1 apresenta um resumo dos métodos que *podem* ser implementados para cada tipo de associação em uma classe, dependendo das necessidades de design.

Observe que na Tabela 10.1 cada tipo de associação tem um diferente conjunto de métodos para acessar e atualizar ligações. A implementação desses métodos segue definições que são usualmente mantidas as mesmas de classe para classe. As subseções seguintes mostram exemplos de alguns desses métodos.

Tabela 10.1 Operações típicas sobre associações dependendo de seu tipo.

Tipo	Para 1	Para 0..1	Para muitos (*)
Get	getPapel():Objeto	getPapel():Objeto	getPapel():Set
Add	Não se aplica	addPapel(obj)	addPapel(obj)
Remove	Não se aplica	removePapel()	removePapel(obj)
Replace	replacePapel(obj)	replacePapel(obj)	replacePapel(antigoObj,novoObj)
Tipo	* {ordered}	* {sequence}	
Get	getPapel():OrderedSet getPapel(posição):Objeto	getPapel():Sequence getPapel(posição):Objeto	
Add	addPapel(posição,obj)	addPapel(posição,obj)	
Remove	removePapel(obj) removePapel(posição)	removePapel(posição)	
Replace	replacePapel(antigoObj,novoObj) replacePapel(posição,obj)	replacePapel(posição,obj)	
Tipo	Mapeamento (qualificador interno)		Mapeamento (qualificador externo)
Get	getPapel():Set getPapel(chave):Objeto		getPapel():Set getPapel(chave):Objeto
Add	addPapel(obj)		addPapel(chave,obj)
Remove	removePapel(obj) removePapel(chave)		removePapel(obj) removePapel(chave)
Replace	replacePapel(antigoObj,novoObj)		replacePapel(antigaChave,novoObj) replacePapel(antigoObj,novoObj)
Get	getPapel():Set getPapel(chave):Set		getPapel():Set getPapel(chave):Set
Add	addPapel(obj)		addPapel(chave,obj)
Remove	removePapel(obj)		removePapel(obj)
Replace	replacePapel(antigoObj,novoObj)		replacePapel(antigoObj,novoObj)
Tipo	* com classe de associação	* {bag}	Array tamanho fixo
Get	getPapel():Set getClasseDeAssociação():Set getClasseDeAssociação(obj):Objeto	getPapel():Bag	getPapel():Array getPapel(posição):Objeto
Add	addPapel(obj)	addPapel(obj)	Não se aplica
Remove	removePapel(obj) removeClasseDeAssociação(obj)	removePapel(obj)	Não se aplica
Replace	replacePapel(antigoObj,novoObj)	Não se aplica	replacePapel(posição,obj)
Tipo	Pilha		Fila
Get	getPapel():Objeto		getPapel():Objeto
Add	pushPapel(obj)		queuePapel(obj)
Remove	popPapel(obj)		removePapel(obj)
Replace	Usualmente não se aplica		Usualmente não se aplica

10.3.1 Associação unidirecional para 1

A associação unidirecional com multiplicidade de papel 1 pode ser armazenada em uma simples variável de instância na classe de origem, e seu tipo deve ser a classe de destino. A Figura 10.3 mostra uma associação unidirecional *para um* de *Carro* para *Pessoa* com nome de papel *dono*.

A implementação da classe *Carro* requer uma variável de instância chamada *dono* com o tipo *Pessoa*. Em relação aos métodos da associação, de acordo com a Tabela 10.1, apenas *getDono* e *replaceDono* deveriam ser implementadas. O pseudocódigo para a classe *Carro*, nesse caso, poderia ser:

```

CLASSE Carro
  VAR ASSOCIAÇÃO PRIVADA dono:Pessoa
  MÉTODO CONSTRUTOR Create (umDono:Pessoa)
    dono:=umDono
  FIM MÉTODO CONSTRUTOR
  MÉTODO getDono():Pessoa
    RETORNA dono
  FIM MÉTODO
  MÉTODO replaceDono (novoDono:Pessoa)
    dono:=novoDono
  FIM MÉTODO
FIM CLASSE

```

No caso de um papel com multiplicidade 0..1, há duas possibilidades:

- Ele pode ser implementado como um conjunto, ou seja, similar ao papel com multiplicidade *. Nesse caso, se não houver objeto ligado, o método *get* deveria retornar o conjunto vazio.
- Ele poderia ser implementado como uma variável simples, ou seja, similar aos papéis com multiplicidade 1. Nesse caso, como mostrado anteriormente, se não houver objeto ligado, o método *get* deveria retornar o objeto *nulo*.

A segunda abordagem é largamente adotada. Existe, inclusive, a possibilidade de usar o padrão de design *objeto nulo* (Woolf, 1998) no lugar do valor *null* fornecido pelas linguagens de programação. A vantagem é que o objeto nulo é independente de linguagem. O objeto nulo é uma instância da classe *Nulo*, cujo comportamento consiste em não fazer nada. Se houver uma iteração sobre um objeto nulo, ela produzirá nada, exatamente como um conjunto vazio. No entanto, iterar sobre o *valor null* ou *nil* poderia produzir uma exceção, o que não seria desejável nesse caso.

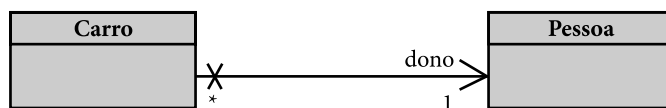


Figura 10.3

Uma classe com associação unidirecional para 1.

Se assumirmos que a multiplicidade de papel na Figura 10.3 é 0..1, existirão pelo menos duas possíveis implementações. A primeira considera a associação como um conjunto:

```
CLASSE Carro
    VAR ASSOCIAÇÃO PRIVADA dono:Set<Pessoa>
    MÉTODO CONSTRUTOR Create()
        dono:=Set.new()
    FIM MÉTODO CONSTRUTOR
    MÉTODO getDono():Set<Pessoa>
        RETORNA dono.protegido()
    FIM MÉTODO
    MÉTODO addDono(novoDono:Pessoa)
        SE dono.size()>0 ENTÃO
            Exception.throw('O carro já tem um dono')
        FIM SE
        dono.add(novoDono)
    FIM MÉTODO
    MÉTODO removeDono()
        SE dono.size()==0 ENTÃO
            Exception.throw('O carro não tem um dono')
        FIM SE
        dono.removeUmElemento()
    FIM MÉTODO
    MÉTODO replaceDono(novoDono:Pessoa)
        SE dono.size()==1 ENTÃO
            dono.removeUmElemento()
        FIM SE
        dono.add(novoDono)
    FIM MÉTODO
FIM CLASSE
```

Toda vez que um método tal como *getDono* retorna um conjunto de objetos, a coleção deve ser protegida contra mudanças. Isso é explicado em mais detalhes adiante neste capítulo.

O método *removeDono* foi implementado pela chamada do método *removeUmElemento* sobre um conjunto. Este método remove um elemento (qualquer um) de um conjunto sem a necessidade de especificar qual. Como o conjunto poderá ter no máximo um elemento neste ponto, não será necessário saber qual elemento deve ser removido do conjunto. O método *removeDono* poderia também ser implementado como *dono:=Set.new()*, com o mesmo resultado final. Porém, como muitos programadores poderiam notar, essa implementação produziria muito lixo na memória, caso objetos fossem adicionados e removidos com frequência naquele papel, e isso poderia eventualmente degradar a performance.

A segunda abordagem considera a associação como uma variável simples, e usa o padrão de design *Objeto nulo*:

```

CLASSE Carro
  VAR ASSOCIAÇÃO PRIVADA dono:Pessoa
  MÉTODO CONSTRUTOR Create()
    dono:=ObjetoNulo.instância()
  FIM MÉTODO CONSTRUTOR
  MÉTODO getDono():Pessoa
    RETORNA dono
  FIM MÉTODO
  MÉTODO addDono(novoDono:Pessoa)
    SE dono<>ObjetoNulo.instância() ENTÃO
      Exception.throw('O carro já tem um dono')
    FIM SE
    dono:=novoDono
  FIM MÉTODO
  MÉTODO removeDono()
    SE dono=ObjetoNulo.instância() ENTÃO
      Exception.throw('O carro não tem um dono')
    FIM SE
    dono:=ObjetoNulo.instância()
  FIM MÉTODO
  MÉTODO replaceDono(novoDono:Pessoa)
    dono:=novoDono
  FIM MÉTODO
FIM CLASSE

```

Ambas as implementações de *replaceDono* mostradas anteriormente assumem que se o dono existir ele é substituído, e que se ele não existir ele é definido. Nenhuma exceção é ativada se *replaceDono* for chamado para um carro que não tem um dono. Nesse caso, ele se comportaria exatamente como *addDono*.

10.3.2 Associação unidirecional para muitos

A associação unidirecional *para muitos* pode ser interpretada como uma estrutura de dados. Se ela for uma simples associação com multiplicidade * no papel destino, então ela pode ser implementada como um conjunto. Adiante segue o pseudocódigo exemplo para a associação representada na Figura 10.4:

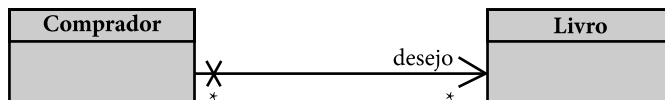


Figura 10.4

Uma classe com uma associação unidirecional simples para muitos.

```
CLASSE Comprador
VAR ASSOCIAÇÃO PRIVADA desejo:SET<Livro>
MÉTODO getDesejo():SET<Livro>
    RETORNA desejo.protegido()
FIM MÉTODO
MÉTODO addDesejo(umLivro:Livro)
    desejo.add(umLivro)
FIM MÉTODO
MÉTODO removeDesejo(umLivro:Livro)
    desejo.remove(umDesejo)
FIM MÉTODO
FIM CLASSE
```

Não é necessário implementar o método *replaceDesejo* aqui porque desejos usualmente são apenas adicionados e removidos de um comprador. Não é uma operação comum substituir um desejo por outro. Assim, este método não foi considerado para implementação.

A mensagem *protegido* enviada para uma coleção produz uma versão não editável dessa coleção para evitar que elementos sejam removidos ou adicionados à coleção original por outros meios que não as mensagens *add* e *remove* que são implementadas na classe.

A implementação dessa proteção pode ser feita de várias formas; algumas delas são específicas de linguagem. Exemplos incluem fazer uma cópia da coleção original (cópia superficial³ ou cópia profunda⁴, dependendo do grau de proteção desejado), ou usar o padrão de design *Procurador protetor*⁵ (Gamma; Helm; Johnson; Vlissides, 1995), o que sugere a implementação de uma classe que encapsula a coleção original e implementa apenas um método para iterar sobre os elementos, mas nenhum método para modificar a coleção original.

Se o papel de associação for rotulado com *{ordered}* ou *{sequence}*, o tipo de dados da variável que representa o papel deve ser trocado pelo tipo de dados correspondente suportado pela linguagem e, adicionalmente, outros métodos específicos devem ser implementados, conforme mencionado na Tabela 10.1.

No caso do papel com limites inferior e superior idênticos, ele pode ser implementado como um *array*. Por exemplo, a multiplicidade 5 poderia ser implementada como um *array* de cinco posições. Os comandos originais *add* e *remove* não são aplicados ao *array*, porque seu tamanho não pode mudar. Entretanto, elementos podem ser substituídos em função de sua posição, como mostrado na Tabela 10.1.

Papéis cuja multiplicidade é um intervalo, como 3..8, podem ser implementados exatamente como papéis para muitos (*). A única diferença é que a coleção deve ter seus limites verificados quando objetos são removidos ou adicionados nela.

³ Apenas ponteiros são copiados, mas os objetos membros da coleção são mantidos os mesmos.

⁴ Tanto os ponteiros quanto os objetos membros e subestruturas são todos copiados.

⁵ *Protection proxy*.

10.3.3 Associação qualificada unidirecional

A associação qualificada unidirecional é implementada de forma similar à associação com multiplicidade para muitos. Entretanto, em vez do tipo de dados *Set*, usa-se uma estrutura de mapeamento, ou dicionário (*MAP*) que associa um tipo alfanumérico, primitivo ou enumeração (chave) com um objeto ou conjunto de objetos (valores).

Assim como para quaisquer outras associações unidirecionais, deve-se manter em mente que a implementação unidirecional só é possível quando a associação não tem limites de multiplicidade na sua origem, ou seja, papel de origem tem multiplicidade *. Se este não for o caso, a implementação bidirecional deve ser considerada.

A Figura 10.5 mostra a implementação de uma associação qualificada definindo um mapeamento para 0..1 com um qualificador interno. A lista de desejos do comprador é agora considerada como uma associação qualificada. O código respectivo é apresentado adiante:

```
CLASSE Comprador
  VAR ASSOCIAÇÃO PRIVADA desejo:MAP<ISBN,Livro>
  MÉTODO getDesejo():SET<Livro>
    RETORNA desejo.getValues()
  FIM MÉTODO
  MÉTODO getDesejo(umISBN:ISBN):Livro
    RETORNA desejo.atKey(umISBN)
  FIM MÉTODO
  MÉTODO addDesejo(umLivro:Livro)
    desejo.add(umLivro.getISBN(),umLivro)
  FIM MÉTODO
  MÉTODO removeDesejo(umLivro:Livro)
    desejo.removeValue(umISBN:ISBN)
  FIM MÉTODO
FIM CLASSE
```

O método *replace* não é implementado para desejos novamente porque não faria sentido substituir um livro por outro.

Nota-se aqui que a estrutura de dados básica *MAP* tem as operações usuais para acessar um valor dada sua chave (*atKey*), acessar o conjunto de todos os valores (*getValues*), incluir um par chave/valor (*add*), remover um par dada sua chave (*removeKey*) ou dado seu valor (*removeValue*), e assim por diante.

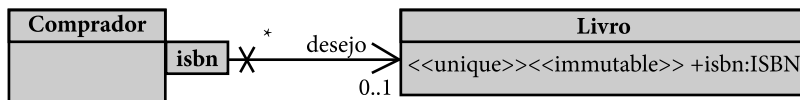
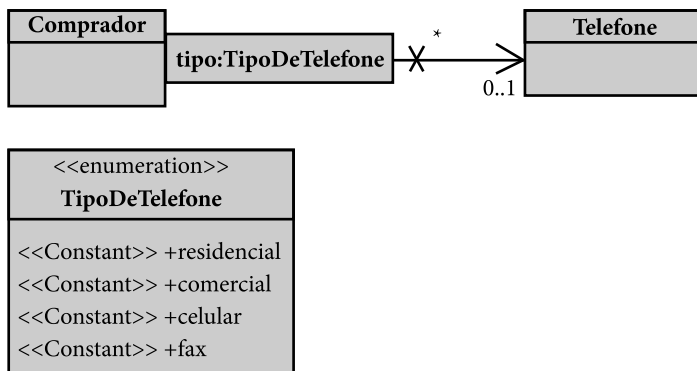


Figura 10.5

Associação qualificada com um qualificador interno.


Figura 10.6

Associação qualificada com um qualificador externo.

É importante enfatizar que o design da associação qualificada com um qualificador interno apenas funciona se o atributo do qualificador for imutável. Se este não for o caso, o atributo poderia mudar e essa mudança não seria necessariamente propagada para o valor que é a chave para a associação. Isso poderia criar uma inconsistência. Essa restrição, porém, não se aplica para mapeamentos com qualificadores externos, porque nesse caso o qualificador não é um atributo da classe qualificada.

A Figura 10.6 apresenta um mapeamento com um qualificador externo. O código correspondente é mostrado a seguir:

```

CLASSE Comprador
  VAR ASSOCIAÇÃO PRIVADA telefone:MAP<TipoDeTelefone, Telefone>
  MÉTODO getTelefone():SET<Telefone>
    RETORNA telefone.getValues()
  FIM MÉTODO
  MÉTODO getTelefone(umTipo:TipoDeTelefone):Telefone
    RETORNA telefone.atKey(umTipo)
  FIM MÉTODO
  MÉTODO addTelefone(umTipo:TipoDeTelefone, umTelefone:Telefone)
    telefone.add(umTipo, umTelefone)
  FIM MÉTODO
  MÉTODO removeTelefone(umTelefone:Telefone)
    telefone.removeValue(umTelefone)
  FIM MÉTODO
  MÉTODO removeTelefone(umTipo:TipoDeTelefone)
    telefone.removeKey(umTipo)
  FIM MÉTODO
FIM CLASSE
  
```

No caso da Figura 10.6 existe uma questão a ser considerada: como a origem da associação não tem restrição de multiplicidade, e o qualificador não é um atributo da classe, nada garante que o mesmo telefone não seja associado a duas ou mais chaves. Por

exemplo, a seguinte sequência de comandos poderia produzir um telefone associado a dois diferentes tipos:

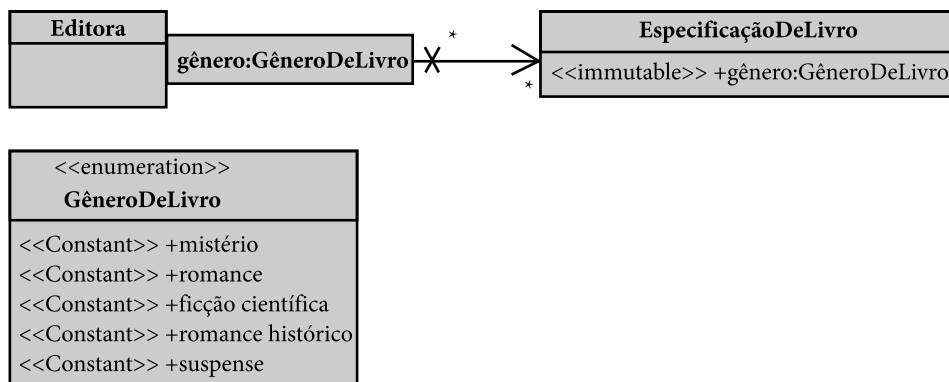
```
umCliente.addTelefone("residencial", umTelefone)
umCliente.addTelefone("comercial", umTelefone)
```

Se isso é o esperado, nada mais precisa ser dito. Mas se isso não é o que se quer representar, o papel no lado esquerdo deverá ser 1 e a associação deverá ser implementada como uma associação bidirecional com mecanismos de controle para evitar que um mesmo telefone seja associado a mais de um tipo.

Para o exemplo seguinte, considera-se que diferentes editoras poderiam editar o mesmo livro. Assim, teremos livros e especificações de livros: um *livro* tem uma única editora, mas uma *especificação de livro* tem um conjunto de editoras. O ISBN do livro para cada editora pode ser diferente, mas a *especificação de livro* é a mesma para diferentes editoras, bem como o gênero do livro. A Figura 10.7 ilustra que uma especificação de livro tem o gênero como único atributo e, portanto, cada especificação de livro tem apenas um gênero que pode ser associado a diferentes editoras. A Figura 10.7 apresenta uma *partição*, ou seja, uma associação qualificada *para muitos*, com um qualificador interno. O código correspondente é mostrado a seguir:

```
CLASSE Editora
    VAR ASSOCIAÇÃO PRIVADA especificaçãoDeLivro:
        RELAÇÃO<GêneroDeLivro, EspecificaçãoDeLivro>
    MÉTODO getEspecificaçãoDeLivro() : SET<EspecificaçãoDeLivro>
        RETORNA especificaçãoDeLivro.getValues()
    FIM MÉTODO
    MÉTODO getEspecificaçãoDeLivro(umGênero: GêneroDeLivro) :
        SET<EspecificaçãoDeLivro>
        RETORNA especificaçãoDeLivro.atKey(umGênero)
    FIM MÉTODO
    MÉTODO addEspecificaçãoDeLivro(umaEspecificaçãoDeLivro:
        EspecificaçãoDeLivro)
        especificaçãoDeLivro.add(umaEspecificaçãoDeLivro.getGênero(),
            umaEspecificaçãoDeLivro)
    FIM MÉTODO
    MÉTODO removeEspecificaçãoDeLivro(umaEspecificaçãoDeLivro:
        EspecificaçãoDeLivro)
        especificaçãoDeLivro.removeValue(umaEspecificaçãoDeLivro)
    FIM MÉTODO
FIM CLASSE
```

A implementação anterior usa uma estrutura de dados chamada *RELAÇÃO*, a qual pode não existir nativamente na maioria das linguagens de programação, mas ela pode ser facilmente implementada. Ela tem uma interface similar a *MAP*, mas permite que a mesma chave seja associada a vários valores, e não apenas um. No exemplo anterior, para cada *GêneroDeLivro*, a estrutura associa um conjunto de instâncias de *EspecificaçãoDeLivro*.


Figura 10.7

Associação qualificada como uma partição com um qualificador interno.

Há uma questão aqui também. O atributo qualificador *gênero* na classe *EspecificaçãoDeLivro* deve ser *imutável* porque ele é um qualificador interno. Se este não for o caso, problemas como os mencionados anteriormente para o mapeamento qualificado poderiam colocar o design em risco. O atributo não é *unique*, porque diferentes especificações de livro podem ter o mesmo gênero.

Pode-se considerar que o design apresentado na Figura 10.7 permite que uma especificação de livro seja ligada a uma editora ou a diferentes editoras com diferentes gêneros? A resposta é *não*. Como o gênero é imutável e o método *addEspecificaçãoDeLivro* toma a chave do atributo *gênero* da *EspecificaçãoDeLivro*, só é possível adicionar uma especificação de livro uma única vez a uma mesma editora. Como o gênero é imutável, a especificação de livro nunca poderia ser adicionada novamente com um gênero diferente, mesmo que em outra editora.

Este não seria o caso se o gênero não fosse imutável: a mesma especificação de livro poderia mudar seu gênero e ser ligada novamente a uma ou mais editoras. Nesse caso, uma ligação inconsistente para a especificação de livro com o antigo gênero seria deixada para trás.

10.3.4 Associação unidirecional com classe de associação

Quando a associação tem uma classe de associação, é necessário implementar a criação e a destruição de instâncias dessa classe cada vez que a ligação correspondente é adicionada ou removida.

Classes de associação podem existir em associações com qualquer multiplicidade. Entretanto, elas são mais comuns e úteis em associações que são de muitos para muitos.

Uma implementação possível para este tipo de associação consiste em criar um *mapeamento* que associa instâncias da classe oposta às instâncias da classe de associação.

A Figura 10.8 mostra uma classe de associação, e sua implementação é vista a seguir:

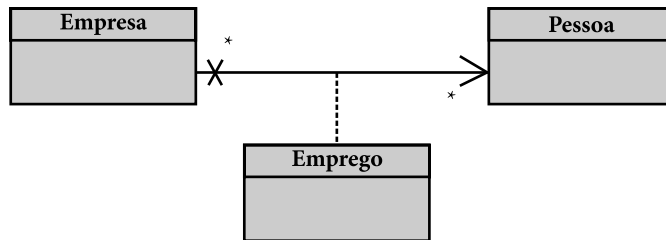


Figura 10.8

Associação unidirecional com classe de associação.

CLASSE Empresa

```

VAR ASSOCIAÇÃO PRIVADA empregado:MAP<Pessoa,Emprego>
MÉTODO getEmpregado():SET<Pessoa>
    RETORNA empregado.getKeys()
FIM MÉTODO
MÉTODO getEmprego(umaPessoa:Pessoa):Emprego
    RETORNA empregado.atKey(umaPessoa)
FIM MÉTODO
MÉTODO addEmpregado(umaPessoa:Pessoa)
    empregado.add(umaPessoa,Emprego.Create())
FIM MÉTODO
MÉTODO removeEmpregado(umaPessoa:Pessoa)
    VAR LOCAL umEmprego:Emprego
    umEmprego:=empregado.atKey(umaPessoa)
    empregado.removeKey(umaPessoa)
    umEmprego.destroy()
FIM MÉTODO
MÉTODO removeEmprego(umEmprego:Emprego)
    empregado.removeValue(umEmprego)
    umEmprego.destroy()
FIM MÉTODO
MÉTODO replaceEmpregado(antigoEmpregado,novoEmpregado:Pessoa)
    self.removeEmpregado(antigoEmpregado:Pessoa)
    self.addEmpregado(novoEmpregado:Pessoa)
FIM MÉTODO
FIM CLASSE
  
```

Na Figura 10.8, vemos que, quando uma nova ligação é criada de *Empresa* para *Pessoa*, uma nova instância de *Emprego* é automaticamente criada.

As operações que explicitamente destroem um emprego quando a ligação é removida são incluídas no código para deixar claro que é isso o que deve ser feito, já que o emprego não pode existir independentemente da ligação que o criou. Em linguagens com coletor de lixo e sem a desalocação explícita de objetos, deve-se

garantir que, após a remoção da ligação, nenhuma outra referência forte⁶ ao objeto seja mantida.

O comando *replace* apenas deleta a antiga ligação e cria uma nova. A instância de *Emprego* não é mantida quando o papel é substituído. Entretanto, em alguns casos, este poderia ser o significado pretendido (substituir o objeto associado mantendo a mesma instância da classe de associação).

10.4 Associações bidirecionais

Como mencionado anteriormente, a implementação unidirecional de associações só é possível quando elas são navegáveis em apenas uma direção e não há restrições de multiplicidade no papel de origem. Em outras situações, a implementação bidirecional é necessária.

Pelo menos três padrões para implementação de associações bidirecionais foram propostos (Fowler, 2003):

- Implementar a associação como duas associações unidirecionais (padrão *Amigos mútuos*).
- Implementar a associação como uma associação unidirecional em apenas uma das classes. A navegação seria possível na direção oposta por meio de uma consulta.
- Implementar um objeto intermediário que representa a associação.

Em todos os três casos anteriores, se a associação é navegável em ambos os sentidos, o método *get* deve ser implementado em ambas as classes participantes. Entretanto, se for o caso de uma associação unidirecional com restrição de multiplicidade na origem, o método *get* só deve ser implementado na classe de origem.

Os métodos *add* e *remove*, se requeridos, devem ser implementados em apenas uma das classes, porque se existissem em ambas as classes eles seriam redundantes, já que eles fariam exatamente a mesma coisa.

10.4.1 Amigos mútuos

A opção de implementar associações bidirecionais nas duas direções é mais eficiente em termos de tempo, mas menos eficiente em termos de alocação de espaço, porque cada associação é implementada duas vezes. Isso também pode requerer mecanismos de controle extras porque a implementação da associação em ambas as classes precisa ser sincronizada.

A implementação dos componentes unidirecionais da associação segue as recomendações dadas da Seção 10.3. Entretanto, três subcasos ainda devem ser considerados aqui:

- Ambos os papéis são opcionais.
- Apenas um papel é obrigatório.
- Ambos os papéis são obrigatórios.

⁶ Uma referência *forte* impediria o coletor de lixo de remover o objeto da memória. Uma referência *fraca* seria uma referência não tomada em conta pelo coletor de lixo, ou seja, um objeto apenas com referências fracas poderia ser desalocado assim mesmo.

10.4.1.1 Ambos papéis opcionais

Se ambos os papéis são opcionais e nenhuma outra restrição está presente, então ligações podem ser adicionadas e removidas a vontade. Como as ligações podem ser adicionadas e removidas de ambos os lados da associação, métodos auxiliares seriam necessários para adicionar e remover as ligações unidirecionais individuais.

Esses métodos auxiliares não devem ser invocados em nenhuma outra parte exceto os métodos *add* e *remove*. É por isso que eles deveriam ser declarados como *privados*, embora a classe oposta deva acessá-los; assim eles são exportados exclusivamente para aquela classe.

A Figura 10.9 apresenta uma associação bidirecional de “muitos para muitos”. O código de implementação correspondente é mostrado a seguir:

```

CLASSE Comprador
  VAR ASSOCIAÇÃO PRIVADA desejo:SET<Livro>
  MÉTODO getDesejo():SET<Livro>
    RETORNA desejo.protegido()
  FIM MÉTODO
  MÉTODO addDesejo(umLivro:Livro)
    desejo.add(umLivro)
    umLivro.privadoAddInteressado(self)
  FIM MÉTODO
  MÉTODO removeDesejo(umLivro:Livro)
    desejo.remove(umLivro)
    umLivro.privadoRemoveInteressado(self)
  FIM MÉTODO
FIM CLASSE

CLASSE Livro
  VAR ASSOCIAÇÃO PRIVADA interessado:SET<Comprador>
  MÉTODO getInteressado():SET<Comprador>
    RETURN interessado.protegido()
  FIM MÉTODO
  MÉTODO PRIVADO privadoAddInteressado(umComprador:Comprador)
  EXPORTADO PARA: Comprador
    interessado.add(umComprador)
  FIM MÉTODO
  MÉTODO PRIVADO privadoRemoveComprador(umComprador:Comprador)
  EXPORTADO PARA: Comprador
    interessado.remove(umComprador)
  FIM MÉTODO
FIM CLASSE

```



Figura 10.9

Associação bidirecional de “muitos para muitos”.

Os métodos auxiliares no código anterior são declarados privados, mas exportados apenas para a classe oposta.

Como já mencionado, os métodos de acesso (*get*) devem ser implementados em ambas as classes se a associação for navegável em ambos os sentidos. Mas os métodos *add* e *remove* podem ser implementados em apenas uma classe. No exemplo anterior, eles foram implementados apenas na classe *Comprador*.

10.4.1.2 Apenas um papel obrigatório

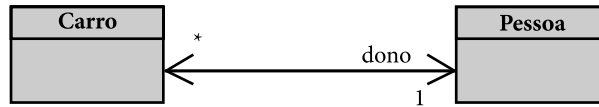
Se um dos papéis for obrigatório, como, por exemplo, 1 para *, então a equipe deve se perguntar se o papel obrigatório é imutável ou não. Por exemplo, um pagamento tem um pedido e não pode mudar para outro pedido (o papel é imutável), mas um carro tem um dono e pode mudar para outro dono, caso seja vendido.

Vamos inicialmente examinar o caso de uma associação que pode ser atualizada, como mostrado na Figura 10.10.

O código para implementar as classes da Figura 10.19 poderia ser o seguinte:

```
CLASSE Carro
    VAR ASSOCIAÇÃO PRIVADA dono:Pessoa
    MÉTODO CONSTRUTOR Create(umDono:Pessoa)
        dono:=umDono
        umDono.privadoAddCarro(self)
    FIM MÉTODO CONSTRUTOR
    MÉTODO getDono():Pessoa
        RETORNA dono
    FIM MÉTODO
    MÉTODO replaceDono(umDono:Pessoa)
        umDono.privadoRemoveCarro(self)
        umDono.privadoAddCarro(self)
        dono:=umDono
    FIM MÉTODO
FIM CLASSE

CLASSE Pessoa
    VAR ASSOCIAÇÃO PRIVADA carros:SET<Carro>
    MÉTODO getCarros():SET<Carro>
        RETORNA carros.protegido()
    FIM MÉTODO
    MÉTODO PRIVADO privadoAddCarro(umCarro:Carro)
    EXPORTADO PARA: Carro
        carros.add(umCarro)
    FIM MÉTODO
    MÉTODO PRIVADO privadoRemoveCarro(umCarro:Carro)
    EXPORTADO PARA: Carro
        carros.remove(umCarro)
    FIM MÉTODO
FIM CLASSE
```

**Figura 10.10**

Associação bidirecional com um papel obrigatório que pode ser atualizado.

Como se pode ver, a classe *Carro* implementa o único método público de atualização, que é *replaceDono*. Uma nova ligação de posse é criada toda vez que uma nova instância de *Carro* é criada, como definido no construtor da classe *Carro*. A partir daí, a posse só pode ser trocada de uma pessoa para outra. A classe *Pessoa* implementa apenas os métodos privados para adicionar e remover um carro de sua coleção local (e o *getter*).

Quando o dono de um carro é substituído, isso é feito em três passos: primeiro, a ligação para o carro é removida do antigo dono, depois uma ligação para o carro é adicionada ao novo dono, e finalmente a ligação do carro para o novo dono é adicionada.

O segundo caso a ser considerado aqui é quando o papel é imutável. Este é o caso de uma associação de 1 para * de *Comprador* para *Pedido*, por exemplo: ela é obrigatória para *Pedido* e um pedido não pode nunca mudar seu cliente. Essa situação é mostrada na Figura 10.11 e o código correspondente segue:

```

CLASSE Pedido
  VAR ASSOCIAÇÃO PRIVADA comprador:Comprador
  MÉTODO CONSTRUTOR Create(umComprador:Comprador)
    comprador:=umComprador
    umComprador.privadoAddPedido(self)
  FIM MÉTODO CONSTRUTOR
  MÉTODO getComprador():Comprador
    RETORNA comprador
  FIM MÉTODO
FIM CLASSE

CLASSE Comprador
  VAR ASSOCIAÇÃO PRIVADA pedido:SET<Pedido>
  MÉTODO getPedido():SET<Pedido>
    RETORNA pedido.protegido()
  FIM MÉTODO
  MÉTODO removePedido(umPedido:Pedido)
    pedido.remove(umPedido)
    umPedido.destroy()
  FIM MÉTODO
  MÉTODO PRIVADO privadoAddPedido(umPedido:Pedido)
  EXPORTADO PARA: Pedido
    pedido.add(umPedido)
  FIM MÉTODO
FIM CLASSE
  
```


Nesse caso, um pedido é imediatamente associado a um comprador quando for criado. Se um pedido é removido de um comprador então ele deve ser destruído.

10.4.1.3 Dois papéis obrigatórios

Há situações nas quais os dois papéis de uma associação são obrigatórios. Nesse caso, nenhum objeto pode existir sem estar ligado ao outro. Isso significa que a criação das ligações deve acontecer no construtor de ambas as classes. A Figura 10.12 mostra um exemplo de uma associação de 1 para 1..*; ela é obrigatória nos dois sentidos.

Mais uma vez, os métodos que serão implementados dependerão de uma decisão sobre se os papéis são imutáveis ou não. O papel do comprador do ponto de vista de um endereço é imutável. Isso significa que um endereço não pode mudar de um comprador para outro⁷.

Quando um comprador é criado, ele deve ter pelo menos um endereço. Isso significa que o criador de um comprador deve receber todos os dados necessários para instanciar esse endereço (apenas *rua* e *número* são mostrados no exemplo, para simplificar).

Um comprador existente pode adicionar um novo endereço, mas o método que adiciona um endereço não pode simplesmente receber uma instância de endereço como um argumento, porque nenhum endereço pode existir sem estar ligado a um comprador. Assim, o comprador deve implementar um método para adicionar um novo endereço que recebe os dados necessários para instanciar tal endereço. O código para implementar essa situação é mostrado a seguir:

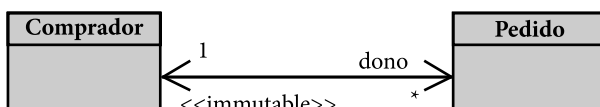


Figura 10.11

Uma associação bidirecional com um papel obrigatório imutável.

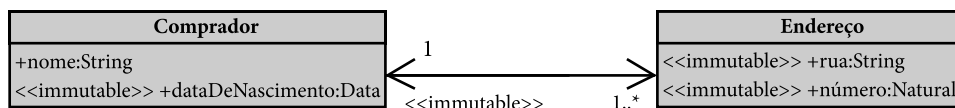


Figura 10.12

Uma associação bidirecional que é obrigatória nos dois sentidos.

⁷ De fato, na vida real, um endereço pode mudar de uma pessoa para outra se considerarmos que uma pessoa pode se mudar e outra pessoa ocupar o mesmo endereço que já havia sido registrado. Entretanto, usualmente, sistemas não tratam endereços de forma tão precisa: pragmaticamente, se um comprador se muda, é sempre um novo endereço que é registrado.

CLASSE Endereço

```

VAR ASSOCIAÇÃO PRIVADA comprador:Comprador
VAR ATRIBUTO PRIVADO rua:String
VAR ATRIBUTO PRIVADO número:Natural
MÉTODO CONSTRUTOR Create(umComprador:Comprador, umaRua:String,
    umNúmero:Natural)
    comprador:=umComprador
    rua:=umaRua
    número:=umNúmero
FIM MÉTODO CONSTRUTOR
MÉTODO getComprador():Comprador
    RETORNA comprador
FIM MÉTODO

```

FIM CLASSE**CLASSE** Comprador

```

VAR ASSOCIAÇÃO PRIVADA endereço:SET<Endereço>
VAR ATRIBUTO PRIVADO nome:String
VAR ATRIBUTO PRIVADO dataDeNascimento:Data
MÉTODO CONSTRUTOR Create(umNome:String,
    umaDataDeNascimento:Data, umaRua:String, umNúmero:Natural)
    nome:=umNome
    dataDeNascimento:=umaDataDeNascimento
    endereço:=Set.new()
    endereço.add(Endereço.Create(self, umaRua, umNúmero))
FIM MÉTODO CONSTRUTOR
MÉTODO getEndereço():SET<Endereço>
    RETORNA endereço.protegido()
FIM MÉTODO
MÉTODO addEndereço(umaRua:String, umNúmero:Natural)
    endereço.add(Endereço.Create(self, umaRua , umNúmero))
FIM MÉTODO
MÉTODO removeEndereço(umEndereço)
    SE endereço.size()>1 ENTÃO
        endereço.remove(umEndereço)
        umEndereço.destroy()
    SENÃO
        Exception.throw('O comprador deve ter pelo menos um endereço')
    FIM SE
FIM MÉTODO
MÉTODO getNome()String
    RETORNA nome
FIM MÉTODO
MÉTODO getDataDeNascimento():Data
    RETORNA dataDeNascimento
FIM MÉTODO
MÉTODO setNome(umNome:String)
    nome:=umNome
FIM MÉTODO
FIM CLASSE

```

Se o papel de *Endereço* para *Comprador* não fosse imutável, então um comando *replaceComprador* poderia ser implementado também na classe *Endereço*. Mas este não é o caso.

10.4.2 Implementação unidirecional

Mesmo que a associação seja bidirecional, pode ser o caso de que a navegação ocorra muito mais frequentemente, ou seja, muito mais crítica em uma direção do que em outra. Se isso acontecer, uma opção é implementar a associação fisicamente em *apenas uma direção*, e implementar o método *get* do lado oposto como uma pesquisa. A vantagem é que o código fica mais simples e mais rápido em uma direção, além de economizar espaço em memória. A desvantagem é que a navegação do lado oposto poderia ficar bem mais lenta.

Essa forma de implementação é também possível apenas quando o papel na origem não tem restrições de multiplicidade. Se houver tal restrição, então a implementação unidirecional só será possível se mecanismos adicionais de controle da multiplicidade na origem da associação forem implementados, com perda de performance. Adiante há um exemplo de implementação unidirecional para a associação da Figura 10.10, na qual a associação é fisicamente implementada apenas na classe *Carro*:

```
CLASSE Carro
    VAR ASSOCIAÇÃO PRIVADA dono:Pessoa
    MÉTODO CONSTRUTOR Create (umDono:Pessoa)
        dono:=umDono
    FIM MÉTODO CONSTRUTOR
    MÉTODO getDono():Pessoa
        RETORNA pessoa
    FIM MÉTODO
    MÉTODO replaceDono (novoDono:Pessoa)
        dono:=novoDono
    FIM MÉTODO
FIM CLASSE
CLASSE Pessoa
    MÉTODO getCarro():SET<Carro>
        VAR LOCAL carros:SET<Carro>
        carros:=Set.new()
        PARA CADA carro EM Carro.getAllInstances() FAÇA
            SE carro.getDono()==self THEN
                carro.add(car)
        FIM SE
    FIM PARA
    RETORNA carros
    FIM MÉTODO
FIM CLASSE
```

A classe *Carro* é implementada exatamente como se a associação fosse unidirecional a partir dela para *Pessoa*. A classe *Pessoa* implementa apenas o método *getCarros* por meio de uma pesquisa no conjunto de todas as instâncias de *Carro*. Essa implementação apenas funciona bem se a multiplicidade de papel na origem não for restrita.

Em relação à complexidade de tempo dos métodos *get*, a implementação bidirecional executa em *tempo constante*⁸ nas duas direções, e a implementação unidirecional tem tempo constante para *getDono* e *tempo linear*⁹ para *getCarros*, isto é, a performance do método *getCarros* depende do número de carros registrado. Este design pode ser otimizado pelo uso das técnicas de *hash* para indexar o conjunto de compradores relativamente aos seus pedidos. Dessa forma, a complexidade da consulta *getCarros* pode se tornar quase constante na prática ao custo de espaço extra de memória.

Uma limitação dessa técnica é que a linguagem de programação deve prover um método para acessar todas as instâncias de uma classe. Caso contrário, o programador deverá providenciar tal mecanismo. Infelizmente, este é um estilo de projeto arriscado que pode causar problemas em razão da visibilidade global dessas instâncias (Cardoso, 2011).

10.4.3 Implementação com um objeto intermediário

Uma associação bidirecional pode também ser implementada por meio de um objeto intermediário que representa a associação. O objeto intermediário consiste de uma tabela com pares de instâncias ligadas. Segue uma possível implementação para a associação bidirecional de muitos para um da Figura 10.11 com um objeto intermediário:

```
VAR GLOBAL pedido_comprador:MAP<Pedido,Comprador>
VISIBILIDADE RESTRITA PARA: Pedido, Comprador
CLASSE Comprador
  MÉTODO getPedido():SET<Order>
    RETORNAR pedido_comprador.getKeysFor(self)
  FIM MÉTODO
  MÉTODO addPedido(umPedido:Pedido)
    pedido_comprador.add(umPedido,self)
  FIM MÉTODO
  MÉTODO removePedido(umPedido:Pedido)
    pedido_comprador.remove(umPedido,self)
  FIM MÉTODO
FIM CLASSE
CLASSE Pedido
  MÉTODO CONSTRUTOR Create(umComprador:Comprador)
    pedido_comprador.add(self,umComprador)
  FIM MÉTODO CONSTRUTOR
  MÉTODO getComprador():Comprador
    RETORNA pedido_comprador.atKey(self)
  FIM MÉTODO
FIM CLASSE
```

⁸ O tempo da consulta permanece igual mesmo se o número de carros aumentar.

⁹ O tempo aumenta à medida que o número de compradores aumenta. Tempo linear significa que o tempo é uma função $t(x)=ax+b$, onde x é o tamanho do conjunto de clientes e a e b são constantes.

Se a linguagem de programação permitir, a associação deveria ser declarada como uma variável global que só é visível pelas classes participantes. Infelizmente, a maioria das linguagens comerciais não permite esse tipo de característica e a associação teria de ser implementada como uma variável global sem restrição de visibilidade.

Se a associação fosse de “muitos para muitos”, em vez de “um para muitos”, então o tipo de dados *MAP* usado antes teria de ser substituído por *RELATION* com pequenos ajustes nos *getters* e *setters*.

A abordagem do objeto intermediário tende a ser muito mais simples e manutenível do que as duas anteriores. Ela também reflete mais diretamente na estrutura relacional de um possível banco de dados já que as associações, como será visto no Capítulo 13, adiante, são implementadas como tabelas intermediárias, correspondendo ao objeto intermediário apresentado aqui.

A desvantagem desse método é que os *getters* ficam mais lentos em ambas as direções quando comparados com os da abordagem de amigos mútuos e a visibilidade global para o objeto intermediário pode originar um design perigoso se cuidados especiais não forem tomados.

10.5 Métodos delegados e operações de sistema

Até este ponto, foi mostrado como gerar código para classes, atributos, associações e seus métodos básicos correspondentes que foram considerados como parte da estrutura básica das classes. Agora é hora de explicar como implementar métodos delegados e operações de sistema. Esses devem ser implementados seguindo os modelos dinâmicos explicados no Capítulo 9.

Operações de sistema são implementadas como uma sequência de mensagens rotuladas com 1, 2, 3, ..., n que são enviadas pelo controlador. Um método delegado rotulado com x no diagrama de interação deveria ser implementado pela sequência de mensagens rotuladas $x.1$, $x.2$, $x.3$, ..., $x.n$ que são enviadas pelo objeto que recebeu a mensagem rotulada como x .

Vamos olhar novamente para a Figura 9.43, reproduzida aqui como Figura 10.13.

O comando de sistema *adicionarAoCarrinho* deveria, portanto, ser implementado como a sequência de mensagens rotuladas com 1, 2 e 3. O pseudocódigo correspondente poderia ser parecido com o seguinte:

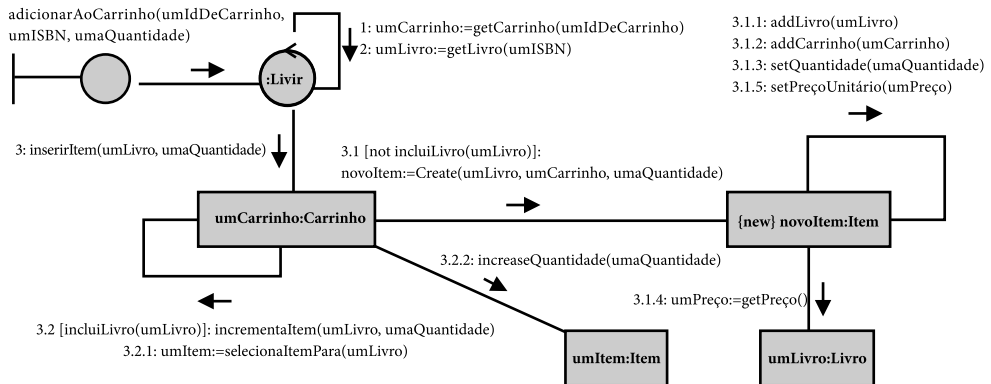


Figura 10.13

Diagrama de comunicação de referência.

```
CLASSE Livro
  VAR ASSOCIAÇÃO carrinho:MAP<IdDeCarrinho,Carrinho>
  VAR ASSOCIAÇÃO livro:MAP<ISBN,Livro>
  MÉTODO adicionarAoCarrinho (umIdDeCarrinho:IdDeCarrinho,
    umISBN:ISBN, umaQuantidade:Natural)
    VAR LOCAL umCarrinho:Carrinho
    VAR LOCAL umLivro:Livro
    umCarrinho:=carrinho.at(umIdDeCarrinho) -- mensagem 1
    umLivro:=livro.at(umISBN) -- mensagem 2
    umCarrinho.inserirLivro(umLivro,umaQuantidade) -- mensagem 3
    ...
  FIM MÉTODO
FIM CLASSE
```

A implementação do método delegado *inserirItem* na classe *Carrinho* consiste da sequência de mensagens 3.1 e 3.2, porque *inserirItem* está rotulada como 3. Essas mensagens são condicionais e mutuamente exclusivas. Sua ordem foi invertida para evitar o uso da negação na condição do “se”. Assim, sua implementação corresponde a uma estrutura “se-então-senão-fimse”, como mostrado adiante:

```
CLASSE Carrinho
...
MÉTODO inserirItem(umLivro:Livro, umaQuantidade:Natural)
    VAR LOCAL novoItem:Item
    SE self.incluiLivro(umLivro) ENTÃO
        self.incrementarItem(umLivro,umaQuantidade) -- 3.2
    SENÃO
        novoItem:=Item.Create(umLivro,self,umaQuantidade) -- 3.1
    FIM SE
FIM MÉTODO
FIM CLASSE
```

Note que o nome de variável *umCarrinho* não é conhecido neste método e deve ser substituído aqui por *self*, que representa a instância de carrinho que está realizando o método.

O método delegado *incrementarItem* na classe *Carrinho* é rotulado com 3.2. Portanto, sua implementação é a sequência de mensagens 3.2.1 e 3.2.2:

```
CLASSE Carrinho
...
MÉTODO incrementarItem(umLivro:Livro, umaQuantidade:Natural)
    VAR LOCAL umItem:Item
    umItem:=self.selecionarItemPara(umLivro) -- 3.2.1
    umItem.increaseQuantidade(umaQuantidade) -- 3.2.2
FIM MÉTODO
FIM CLASSE
```

O construtor da classe *Item* é complexo e como é rotulado com 3.1, sua implementação consiste da sequência de mensagens 3.1.1 a 3.1.5. A classe *Item* poderia se parecer com isso:

```
CLASSE Item
MÉTODO CONSTRUTOR Create(umLivro:Livro, umCarrinho:Carrinho,
    umaQuantidade:Natural)
    VAR LOCAL umPreço:Moeda
    self.addLivro(umLivro) -- 3.1.1
    self.addCarrinho(umCarrinho) -- 3.1.2
    self.setQuantidade(umaQuantidade) -- 3.1.3
    umPreço:=umLivro.getPreço() -- 3.1.4
    self.setPreço(umPreço) -- 3.1.5
FIM MÉTODO CONSTRUTOR
...
FIM CLASSE
```

Os outros métodos que aparecem na Figura 10.13 são básicos e devem ser implementados seguindo sua definição padrão apresentada anteriormente.

10.6 Padrões para consultas filtradas

Nos exemplos mostrados até aqui, basicamente dois tipos de mensagens *get* foram implementadas: as que retornam todos os objetos de um papel e as que retornam um único objeto dada sua identificação, tal como um qualificador ou posição.

Entretanto, algumas vezes, será necessário obter um subconjunto de objetos em um dado papel. Tal subconjunto é usualmente obtido por um filtro como “compradores acima de 30 anos”, “pedidos entre R\$ 100,00 e 200,00”, “pedidos acima de R\$ 100,00 emitidos na semana passada” etc.

Existem pelo menos três padrões para lidar com essa diversidade de consultas (Fowler, 2003):

- Implementar uma única consulta que retorna todos os objetos e deixa para o objeto que necessita a informação a responsabilidade de aplicar um filtro a esta coleção.

- Implementar *consultas específicas*, cada uma com um filtro diferente, de forma que o objeto que necessita da informação possa chamar um método específico em cada caso.
- Implementar uma consulta genérica que use um *objeto filtro*.

As abordagens da consulta única e do objeto filtro implicam a implementação de um único método que deixa mais simples a classe que detém a responsabilidade. Entretanto, elas requerem que mais código seja escrito nas classes que necessitam da informação.

A abordagem das consultas específicas tem como consequência um grande número de métodos na classe que representa a responsabilidade. Mas as classes que necessitam da informação poderiam fazer chamadas simples e receber a informação sem qualquer necessidade de pós-filtragem.

A escolha por um padrão ou outro deve ser feita pelo designer dependendo do número de filtros possíveis e chamadas potenciais. Se existem apenas poucas possibilidades para filtragem e muitas chamadas, a melhor escolha é a abordagem das consultas específicas. Se a quantidade de filtros é grande e existem poucas chamadas para cada filtro, então a consulta única ou objeto filtro poderiam ser melhores. A consulta única é mais simples, mas produz mais trabalho durante a atividade de codificação: cada vez que um filtro é usado, o código respectivo deve ser implementado na classe que chama a consulta. A abordagem de objeto filtro requer a definição de uma classe que implementa o *objeto filtro*, mas após este investimento inicial a abordagem do objeto filtro tende a ser mais simples do que a consulta única.

O objeto filtro é um parâmetro que é passado para o método de consulta genérico. O objeto filtro deve conter atributos e associações para outros objetos. Esses atributos e associações são usados pela consulta para decidir quais objetos devem ser retornados.

Por exemplo, vamos revisar a definição da classe *Livro* mostrada na Figura 10.14.

Suponha agora que necessitamos consultar livros com base em filtros dados por *título*, *nomeDoAutor* ou *preço*.

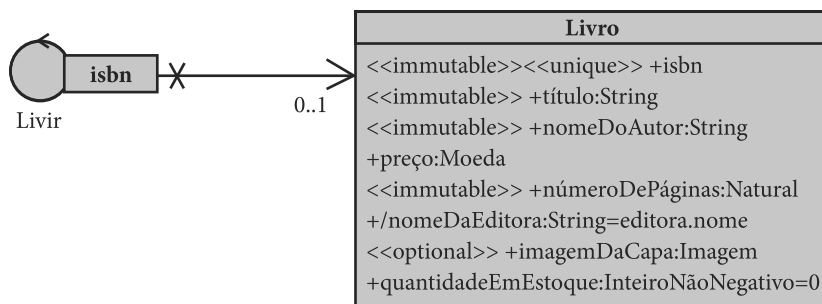


Figura 10.14

Classe de referência para consultas.


Figura 10.15

Uma classe para um objeto filtro.

Aplicando a abordagem da consulta única, um método *getLivros()* poderia ser implementado pela classe *Livir* que é responsável por retornar o conjunto de todos os livros. Se outro objeto necessitar filtrar este conjunto, ele deve aplicar o filtro ao conjunto resultante.

Se a abordagem das consultas específicas for usada, a classe *Livir* deveria implementar três consultas que seriam variações da consulta base:

- *getLivroPorTítulo(umTítulo:String)*.
- *getLivroPorNomeDoAutor(umNomeDeAutor:String)*.
- *getLivroPorPreço(umIntervalo:Intervalo<Moeda>)*.

Cada uma dessas consultas retorna apenas os elementos que satisfazem os respectivos filtros indicados como argumentos. Se uma combinação de critérios for necessária, então novas consultas deveriam ser criadas para os critérios combinados.

A abordagem do objeto filtro requer a definição de uma classe para representar o objeto filtro. Para o exemplo, a classe *FiltroLivro* poderia ser definida como mostrado na Figura 10.15.

Nesse caso, a classe *Livir* deveria implementar um único método *getLivro(umFiltroLivros:Set<Livro>)* para consultar o conjunto de todos os livros.

Usando este método, uma consulta para retornar os livros escritos por “Douglas N. Adams” que custam menos de R\$ 10,00 poderia ser escrita assim:

```
...
meuFiltro:=FiltroLivro.Create()
meuFiltro.setNomeDoAutor("Douglas N. Adams")
meuFiltro.setIntervaloDePreço(Intervalo(R$0,R$10))
livrosBaratosDeAdams:=self.getLivros(meuFiltro)
...
```

Como resultado, a variável *livrosBaratosDeAdams* conteria o conjunto de todas as instâncias de *Livro* cujo autor é “Douglas N. Adams” e que custam menos de R\$ 10,00. Como o atributo *título* de *FiltroLivro* não foi preenchido para a consulta, ele é ignorado e qualquer título poderia se qualificar para o filtro.

10.7 O processo visto aqui

	Concepção	Elaboração
Modelagem de negócio		
Requisitos		
Análise e Design		
Implementação		Gerar código: • Para classes. • Para atributos. • Para associações. • Necessário para métodos básicos como get, set, add, remove, Create, destroy e outros. • Para operações de sistema e métodos delegados usando os modelos dinâmicos como referência.
Teste		
Gerenciamento de Projeto		

10.8 Questões

1. Explique em detalhes como uma mensagem delegada que aparece em um diagrama de comunicação deve ser implementada.
2. Quais são as operações básicas que devem ser implementadas para a maioria dos tipos de associação? Quais devem ser implementadas apenas para tipos específicos de associações?
3. Proponha uma implementação genérica para a consulta *getLivros* (*umFiltroLivro:FiltroLivro*):*Set<Livro>*. Tente mantê-la o mais geral e reusável possível.
4. Proponha uma implementação para os comandos de sistema e métodos delegados apresentados nas Figuras 9.42 e 9.44.

Teste

11

TÓPICOS-CHAVE NESTE CAPÍTULO

- Teste funcional
- *Stubs e drivers*
- Desenvolvimento orientado a teste
- Teste de unidade
- Teste de operações de sistema
- Teste de casos de uso

11.1 Introdução ao teste

Não importa quão sofisticadas sejam as técnicas de modelagem e especificação usadas para desenvolver software, e não importa quão disciplinada e competente seja a equipe, existe um fator que faz com que o teste de software seja sempre necessário: o *erro humano*. É um mito pensar que bons desenvolvedores trabalhando com ferramentas do estado da arte são capazes de desenvolver software livre de erros (Beizer, 1990).

A Lei de Murphy (Bloch, 1980), em vários de seus corolários, parece falar diretamente para a indústria de software. Por exemplo:

- Tudo o que pode dar errado dá errado.
- Se alguma coisa parece estar dando certo, é porque você não olhou direito.
- A natureza sempre favorece a falha oculta.

Por muitos anos, as atividades de teste de software foram consideradas uma punição para os programadores. O teste era considerado uma perda de tempo porque se imaginava que o software deveria ter sido construído corretamente desde o início.

Entretanto, as coisas mudaram. A disciplina de teste agora é considerada extremamente importante. Atualmente, ela é parte integrante dos processos de desenvolvimento de software e uma das disciplinas do Processo Unificado.

Além disso, grandes empresas de software começaram a terceirizar o teste de software, pela contratação de *fábricas de teste*. Isso significa que não apenas desenvolvedores, mas equipes especialmente treinadas estão conduzindo as atividades de teste.

Este capítulo apresenta atividades de teste que são fortemente adaptadas para uso com as técnicas mostradas nos capítulos anteriores.

Testes de unidade são usados basicamente para verificar o comportamento de classes, incluindo seus métodos básicos e delegados. Se geração automática de código for usada,

os testes poderiam ser suprimidos, porque se assume que os geradores automáticos de código não cometem erros humanos.

Testes de operação de sistema são testes funcionais de alto nível que podem ser baseados no exame dos contratos de operação de sistema: seus parâmetros e exceções determinam conjuntos de valores válidos e inválidos para os parâmetros que podem ser sistematicamente verificados e as pós-condições determinam os resultados que devem ser obtidos quando entradas válidas são testadas.

Finalmente, *testes de sistema* são realizados de forma sistemática como *testes de casos de uso*. Cada caso de uso é um script que estabelece fluxos normais e alternativos para realizar as metas de negócio. Testes de casos de uso podem ser executados manualmente ou podem ser automatizados. Se o cliente executar os testes de caso de uso, eles são chamados de *testes de aceitação*.

Existem duas grandes categorias de técnicas de teste:

- *Testes estruturais*, os quais avaliam a estrutura interna do código.
- *Testes funcionais*, os quais avaliam operações baseadas apenas nas suas entradas e saídas.

Neste livro, apenas as técnicas funcionais são aplicadas especificamente aos três níveis de teste que foram introduzidos:

- Teste funcional de unidade para operações básicas e delegadas.
- Teste funcional de operações de sistema baseados em seus contratos.
- Testes de sistema e de aceitação baseados em casos de uso de sistema.

Leitores interessados em entender o teste automatizado em mais detalhes podem dar uma olhada no livro de Meszaros (2007), o qual apresenta um conjunto significativo de padrões de teste no estilo *xUnit*, uma família de frameworks para inúmeras linguagens baseado no design original criado por Kent Beck (1989).

11.2 Teste funcional

O teste funcional consiste em uma sequência de testes que define valores de entrada para uma operação e que observa se o resultado foi o esperado. Testes funcionais podem ser realizados sem nenhum conhecimento do código de programação que implementa a operação; apenas seu comportamento é observado. A quantidade de testes a ser conduzidos para verificar se a operação está correta pode ser virtualmente infinito. Mas o teste funcional pode usar técnicas para reduzir o número de testes necessários sem perder cobertura. As técnicas mais úteis para atingir este objetivo são o *particionamento de equivalência* e a *análise de valor limite*, que são explicados nas subseções seguintes.

11.2.1 Particionamento de equivalência

Um dos princípios do teste funcional é a identificação de situações equivalentes. Por exemplo, se uma operação aceita um conjunto de dados (normalidade) e rejeita outro conjunto

(exceção), então pode ser dito que existem dois *conjuntos equivalentes*¹ de dados de entrada para essa operação: *valores aceitos* e *rejeitados*. É usualmente impossível testar todos os valores incluídos nesses conjuntos, porque eles podem ser virtualmente infinitos. Assim, a técnica do *particionamento de equivalência* indica que ao menos um elemento em cada conjunto de equivalência deve ser testado (Burnstein, 2003).

Classicamente, a técnica de particionamento de equivalência considera a divisão das entradas de acordo com os seguintes critérios (Myers; Sandler; Badgett; Thomas, 2004):

- Se os valores válidos estão especificados como um *intervalo* (por exemplo, de 10 a 20), então se define um conjunto válido (10 a 20) e dois conjuntos inválidos (menos do que 10 e mais do que 20).
- Se os valores válidos são especificados como uma *quantidade de valores* (por exemplo, uma lista com cinco elementos), então se define um conjunto válido (lista com cinco elementos) e dois conjuntos inválidos (listas com menos de cinco e listas com mais do que cinco elementos).
- Se as entradas válidas são especificadas como um *conjunto de valores aceitáveis* que podem ser processados de diferentes formas (por exemplo, as *strings* de uma enumeração como “masculino” e “feminino”), então se define um conjunto válido para cada uma das operações válidas e um conjunto inválido para qualquer outro valor.
- Se os valores válidos são especificados por uma condição lógica (por exemplo, “a data final deve ser posterior à data inicial”), então se define um conjunto válido (quando a condição é verdadeira) e um conjunto inválido (quando a condição é falsa).

Os conjuntos de entradas válidas podem ser definidos não apenas em termos de restrições nos dados de entrada, mas também em termos dos resultados que podem ser produzidos. Se a operação que está sendo testada tem comportamento diferente dependendo do valor da entrada, então diferentes conjuntos válidos devem ser definidos para cada comportamento possível. Por exemplo, considere uma operação simples *metade(x:Integer):Integer*. Esta operação aceita que *x* possa ser qualquer inteiro positivo: ela é definida de forma que não possa aceitar zero ou números negativos. Finalmente, considere que a operação produz $(x-1)/2$ para números ímpares e $x/2$ para números pares. Assim, deve-se considerar que *metade* tem três conjuntos de equivalência para o parâmetro *x*:

- Um conjunto válido composto dos inteiros positivos ímpares: 1, 3, 5, ...
- Um conjunto válido composto dos inteiros positivos pares: 2, 4, 6, ...
- Um conjunto inválido composto dos inteiros não positivos: 0, -1, -2, -3, ...

Os valores dos conjuntos de equivalência devem ser sempre restritos ao domínio definido pelo tipo do parâmetro. Por exemplo, se o tipo de parâmetro fosse diferente de *Integer*, então os conjuntos de equivalência teriam de ser redefinidos. Se a operação fosse *metade(x:Natural):Integer*, então os conjuntos de equivalência deveriam ser redefinidos

¹ Embora a literatura normalmente use o termo *classes de equivalência*, que é um conceito matemático, neste livro, o termo *conjuntos de equivalência* é usado para evitar possíveis confusões com o conceito de classes de orientação a objetos.

de forma a ficarem contidos dentro do domínio dos naturais, o qual não include o zero nem os números negativos:

- Um conjunto válido composto dos inteiros positivos ímpares: 1, 3, 5, ...
- Um conjunto válido composto dos inteiros positivos pares: 2, 4, 6, ...
- Nenhum conjunto inválido.

No entanto, assumindo que a operação fosse definida com um tipo e parâmetro mais amplo, como, por exemplo, *metade(x:Real):Integer*, então os conjuntos de equivalência seriam:

- Um conjunto válido composto dos inteiros positivos ímpares: 1, 3, 5, ...
- Um conjunto válido composto dos inteiros positivos pares: 2, 4, 6, ...
- Um conjunto inválido composto dos números não naturais: R-N.

Pode-se ver pelos exemplos anteriores que se o designer restringir o tipo do parâmetro ao máximo possível, então menos conjuntos de equivalência ou conjuntos de equivalência menores serão definidos. É por isso que exemplos em capítulos anteriores usaram o tipo *Natural* em vez de *Integer* sempre que o zero ou os números negativos não podiam ser aceitos. E é por isso que enumerações devem ser usadas em vez de *strings* irrestritas sempre que possível.

11.2.2 Análise de valor limite

Pessoas que trabalham com teste de software costumam dizer que os *bugs*² se escondem nas frestas. Em razão disso, a técnica de particionamento de equivalência é normalmente usada em conjunto com outra técnica conhecida como *análise de valor limite*.

A análise de valor limite consiste em não escolher um valor aleatoriamente dentro de um conjunto de equivalência, mas em escolher dois ou mais valores limites se eles puderem ser determinados.

Em domínios ordenados, tais como números, este critério pode ser aplicado. Por exemplo, se uma operação requer um parâmetro inteiro que é válido apenas se estiver incluído no intervalo [10..20], então existem três conjuntos de equivalência:

- Inválido para qualquer $x < 10$.
- Válido para $x \geq 10$ e $x \leq 20$.
- Inválido para qualquer $x > 20$.

A análise de valor limite sugere que erros eventuais na lógica de um programa não ficam em pontos arbitrários dentro desses intervalos, mas nos pontos limítrofes onde dois intervalos se encontram. Assim, se o domínio do parâmetro é *Integer*:

- Para o primeiro conjunto inválido, o valor 9 deve ser testado.
- Para o conjunto válido, 10 e 20 devem ser testados.
- Para o segundo conjunto inválido, o valor 21 deve ser testado.

² “Insetos” em inglês, mas também pode significar “defeitos no software”.

Assim, se houver um erro de lógica na operação para algum desses valores de entrada, é muito mais provável que ele seja encontrado dessa forma do que se um valor aleatório for escolhido dentro de cada intervalo que define um conjunto de equivalência.

11.3 Stubs e drivers

Frequentemente, partes do software devem ser testadas separadamente do corpo principal de código, mas ao mesmo tempo elas devem se comunicar com outras partes.

Quando um componente *A* que vai ser testado chama operações de um componente *B* que ainda não foi implementado, este último pode ser trocado por uma versão simplificada dele que implementa apenas o comportamento que é absolutamente necessário para realizar o teste do componente *A*. Esta implementação simplificada usada no lugar de um componente que ainda não foi implementado é chamada de *stub*.

Por exemplo, suponha que a classe *A* necessita de um gerador de números primos *B* que ainda não foi implementado. O *n*-ésimo número primo poderia ser gerado por uma função *primo*(*n:Natural*):*Natural*. Uma versão simplificada de *B* pode ser implementada apenas para permitir o teste da classe *A*. Suponha que os cinco primeiros números primos sejam suficientes para testar adequadamente a classe *A*. Assim, um *stub* para *B* poderia ser implementado da seguinte forma:

```
CLASSE B
  MÉTODO STUB primo(n:Natural):Natural
  CASO n SEJA
    1:RETORNE 2
    2:RETORNE 3
    3:RETORNE 5
    4:RETORNE 7
    5:RETORNE 11
  CASO CONTRÁRIO Exception.throw('A implementação do stub
    requer argumento menor ou igual a 5')
  FIM CASO
FIM MÉTODO STUB
FIM CLASSE
```

Dessa forma, a classe *A* pode ser testada sem que se invista tempo na implementação de uma função mais complexa na classe *B*.

No entanto, operações implementadas na classe *A* serão invocadas por outros componentes do sistema. Mas quando a classe *A* está sendo testada, estes componentes podem não existir. Além disso, se quisermos testar a classe *A*, usualmente desejamos que o teste ocorra de uma forma sistemática e reproduzível. Para assegurar isso, um *driver de teste* para a classe *A* pode ser implementado. O *driver* é um componente de código (possivelmente uma nova classe) que tem como único objetivo testar sistematicamente outro componente.

Para a classe *A*, um *driver* poderia ser implementado como uma nova classe chamada *DriverParaA*, por exemplo, como mostrado na Seção 11.5.

Assim, *stubs* e *drivers* são implementações simples que simulam o comportamento de outros componentes. O *stub* é usado no lugar de um componente que poderia ser chamado, mas que não é ainda implementado. O *driver* é usado no lugar de um componente que poderia chamar o componente a ser testado de uma forma sistemática e reproduzível.

Stubs são usualmente *código descartável*. Quando o componente real for implementado, o *stub* não será mais necessário e poderá ser descartado.

No entanto, *drivers* são peças importantes de código que devem ser mantidas para sempre. Toda vez que um componente tiver de ser testado – e isso acontece com frequência durante o desenvolvimento e evolução do software –, o *driver* estaria disponível para realizar o teste automaticamente.

11.4 Desenvolvimento orientado a teste

O *desenvolvimento orientado a teste* ou *TDD*³ (Beck, 2003) é uma técnica e filosofia de programação que incorpora o teste automático ao processo de produção de código. Ele funciona assim:

1. Primeiramente, o programador que recebe a especificação de uma nova funcionalidade que deve ser implementada deveria criar um conjunto de testes automáticos para o código que ainda não existe.
2. Este conjunto de testes deve ser executado e deve-se observar ele falhar. Isso é feito para mostrar que os testes não terão sucesso a não ser que uma nova característica seja implementada no código. Se o teste passar neste ponto, há duas explicações: ou o teste foi mal escrito ou a característica que está sendo testada já está disponível no código, supondo que possa ser uma atualização de código existente que está sendo produzido e não código novo produzido do zero.
3. Então, o código deve ser desenvolvido com o único objetivo de passar nos testes. Se qualquer outra nova funcionalidade for detectada, então o teste deve ser atualizado antes que ela seja introduzida no código. Nesse caso, o processo reinicia do passo 1.
4. Após o código passar em todos os testes, deve ser limpo e melhorado, se necessário, com o objetivo de satisfazer padrões internos de qualidade. Após passar nos testes finais, é considerado estável e pode ser integrado no corpo principal de código do sistema.

Os contratos que foram desenvolvidos no Capítulo 8 e os modelos dinâmicos do Capítulo 9 são fontes excepcionais de informação para o desenvolvimento de testes completos e consistentes antes de se produzir qualquer código, conforme explicado nas próximas seções.

A motivação para o TDD é incentivar o programador a manter o código simples e produzir um patrimônio de testes que permita que qualquer parte do sistema seja automaticamente testada.

³ *Test-Driven Development*.

11.5 Teste de unidade

Testes de unidade são os testes mais elementares e consistem em verificar se um componente individual (unidade) do software foi implementado corretamente. Este componente poderia ser um método complexo ou uma classe inteira. Usualmente, esta unidade ainda estará desconectada do sistema e precisará ser integrada posteriormente.

Testes de unidade são usualmente realizados pelo programador e não pela equipe de teste. TDD requer que o programador desenvolva o código de teste *antes* de escrever o código a ser testado. Este código de teste é um *driver* que deve sistematicamente gerar o conjunto de dados necessário para testar todos os conjuntos de equivalência válidos e inválidos, aplicando a análise de valor limite quando possível.

Um exemplo de teste de unidade consiste em verificar se um método foi corretamente implementado em uma classe. Dê uma olhada novamente na Figura 10.13 e no método delegado 3.2, *incrementarItem(umLivro:Livro, umaQuantidade:Natural)*, o qual deve ser implementado na classe *Carrinho*.

Agora vamos considerar quais são os conjuntos de equivalência para esses parâmetros. Primeiramente, vamos considerar o parâmetro *umLivro:Livro*. Como o tipo de parâmetro é *Livro*, pode-se assumir que nenhum outro tipo de objeto pode ser passado como argumento. Por exemplo, esta operação nunca poderia receber uma instância de *Comprador* ou *Envio* em vez de uma de *Livro*⁴ porque os mecanismos de tipagem da linguagem de programação não permitiriam isso. A equipe deve se preocupar se objetos que *poderiam* ser recebidos aqui seriam válidos ou não.

Se os mecanismos da linguagem não puderem impedir que um método receba um valor nulo como argumento, então o caso *null* sempre teria de ser considerado como um conjunto de equivalência (usualmente inválido) para qualquer parâmetro que seja tipado com um nome de classe.

O método *incrementarItem* foi definido com o seguinte objetivo: incrementar a quantidade de um item que já está no carrinho. Podemos começar considerando que há três conjuntos de equivalência para o parâmetro *umLivro*:

- Um conjunto de equivalência válido, que contém instâncias de *Livro* que estão ligadas a um item em *umCarrinho*.
- Um conjunto de equivalência inválido, que contém instâncias de *Livro* que não estão ligadas a qualquer item de *umCarrinho*.
- Um conjunto de equivalência inválido, que contém apenas o valor *null*.

O segundo parâmetro de *incrementarItem* é um número natural identificado como *umaQuantidade*. A operação assume que a quantidade existente de um item que já está no pedido será incrementada por *umaQuantidade*. O tipo *Natural* não inclui o zero e assim quantidades potencialmente inválidas não são permitidas como argumento. Entretanto, solicitar um número de cópias de um livro que exceda a quantidade em estoque poderia ser considerado uma condição de erro, exceto se o sistema permitisse solicitar

⁴ A menos que elas fossem subclasses de *Livro*, o que não é o caso.

livros que ainda não estão disponíveis. Assumindo que a operação de sistema *adicionarAoCarrinho* que chama *incrementarItem* não definiu como pré-condição que a quantidade solicitada no pedido está disponível no estoque (veja seu contrato na próxima seção), então os seguintes conjuntos de equivalência podem ser considerados para o parâmetro *umaQuantidade:Natural*:

- Um conjunto de equivalência válido que inclui valores naturais que, adicionados à quantidade do respectivo item, produz um resultado que é menor ou igual ao atributo *quantidadeEmEstoque* de *umLivro*.
- Um conjunto de equivalência inválido que inclui todos os valores naturais que adicionados à quantidade do respectivo item são maiores do que o atributo *quantidadeEmEstoque* de *umLivro*.

Os casos de teste potenciais para *incrementarItem* são combinações de conjuntos de equivalência para os dois parâmetros, como mostrado na Tabela 11.1.

Há apenas uma condição de sucesso, a qual é obtida quando valores válidos são obtidos para ambos os parâmetros. Isso acontece quando *umLivro* está ligado a um item do carrinho e *umaQuantidade* mais a quantidade corrente é menor ou igual à quantidade em estoque. Em OCL, esta condição pode ser expressa no contexto de *umCarrinho* como:

```
self.item.livro->includes(umLivro) and
self.item->select(livro=umLivro).quantidade+umaQuantidade <=
umLivro.quantidadeEmEstoque
```

Embora existam cinco combinações na Tabela 11.1 que produzem falha, apenas três podem ser testadas, porque se o livro não estiver ligado a nenhum item ou se o livro for nulo, o valor de *umaQuantidade* não pode ser determinado como válido ou inválido porque não há quantidade corrente para ser comparada com ele. Usualmente, as regras para obter combinações de teste são:

- Tomar todas as combinações de sucesso e definir um caso de teste para cada uma delas.

Tabela 11.1 Combinações dos conjuntos de equivalência para *incrementarItem*

		<i>umLivro</i>		
		<i>umLivro</i> está ligado a <i>umItem</i> no carrinho (válido)	<i>umLivro</i> não está ligado a <i>umItem</i> no carrinho (inválido)	<i>umLivro</i> é null (inválido)
<i>umaQuantidade</i>	<i>umaQuantidade</i> mais a quantidade corrente é menor ou igual ao estoque (válido)	Sucesso	Exceção	Exceção
	<i>umaQuantidade</i> mais a quantidade corrente é maior do que a quantidade em estoque (inválido)	Exceção	Exceção (não pode ser testado)	Exceção (não pode ser testado)

- Tomar um conjunto inválido de cada vez. Mesmo se a operação tiver n parâmetros, considere apenas um conjunto inválido para apenas um parâmetro de cada vez. Os outros parâmetros devem ser válidos ou ser impossível determinar sua validade.
- Apenas tome uma combinação com mais de um conjunto inválido se essa combinação for possível e útil.

Portanto, existem três testes de falha que precisam ser realizados para determinar como *incrementarItem* se comporta com parâmetros inválidos. A primeira acontece quando *umLivro* não está em *umCarrinho*. A segunda acontece quando *umLivro* é nulo. A terceira acontece quando a quantidade desejada ultrapassa a quantidade em estoque.

```
(1) not self.item.livro->includes(umLivro)
(2) umLivro.isNull()
(3) self.item->select(livro=umLivro).quantidade+umaQuantidade >
    umLivro.quantidadeEmEstoque
```

Para evitar testes que deixem efeitos colaterais no sistema, eles são usualmente feitos em três etapas:

1. *Preparar o cenário para o teste.* Usualmente, objetos especiais, chamados *fixtures*, são criados com o único objetivo de permitir que o teste seja feito.
2. *Executar o teste.*
3. *Limpar o ambiente.* O sistema deve retornar ao estado em que estava anteriormente ao teste. *Fixtures* devem ser eliminadas neste ponto.

Vamos considerar primeiro o cenário de teste de sucesso para *incrementarItem*. Qualquer *fixture* criada para este teste deveria resultar verdadeira para a condição de sucesso mencionada acima. Um exemplo de código que gera tais *fixtures* é o seguinte:

CLASSE DriverParaCarrinho

```
MÉTODO testeSucessoIncrementarItem ()
    VAR FIXTURE umLivro:Livro
    VAR FIXTURE umCarrinho:Carrinho
    VAR FIXTURE umItem:Item
    umLivro:=Livro.Create('0752201360', 'Bring me the head of Willy
    the mailboy!', 'Scott Adams', US$12,30, 128, 5)
    -- a quantidade em estoque é 5.
    umCarrinho:=Carrinho.Create()
    -- o id do carrinho é atribuído internamente.
    umItem:=Item.Create(umLivro, umCarrinho, 3)
    ...
```

Como vamos testar a classe *Carrinho*, é possível que alguns de seus métodos ou métodos de outras classes, como *Livro* e *Item*, ainda não estejam implementados. Neste caso, *stubs* poderiam ser usados aqui para permitir que o teste de *incrementarItem* seja feito.

Após o código de inicialização, o teste propriamente dito pode ser feito:

```
...
umCarrinho.incrementarItem(umLivro, 2)
-- usando análise de valor limite
SE umItem.getQuantidade()=5 ENTÃO
    ESCRIVA('incrementarItem - teste de sucesso: passou')
```

SENÃO

 ESCREVA('incrementarItem - teste de sucesso: falhou')

FIM SE

...

Finalmente, devemos descartar as *fixtures*:

...

umItem.destroy()

umCarrinho.destroy()

umLivro.destroy()

FIM MÉTODO

Agora, as condições de falha devem ser testadas também. Aqui, elas são implementadas como três métodos independentes na classe *DriverParaCarrinho*:

MÉTODO testeFalha1IncrementarItem()

-- not self.item.livro->includes(umLivro)

VAR FIXTURE umLivro:Livro

VAR FIXTURE umCarrinho:Carrinho

umLivro:=Livro.Create('0752201360', 'Bring me the head of Willy
the mailboy!', 'Scott Adams', US\$12,30, 128, 5)

umCarrinho:=Carrinho.Create()

-- o livro não está no carrinho

TENTE

 umCarrinho.incrementarItem(umLivro,2)

 ESCREVA('incrementarItem - teste de falha 1: falhou')

CAPTURE Exception.itemAusente

 ESCREVA('incrementarItem - teste de falha 1: passou')

FIM TENTE

umCarrinho.destroy()

umLivro.destroy()

FIM MÉTODO

MÉTODO testeFalha2IncrementarItem()

-- umLivro.isNull()

VAR FIXTURE umLivro:Livro

VAR FIXTURE umCarrinho:Carrinho

umCarrinho:=Carrinho.Create() -- o livro é nulo

TENTE

 umCarrinho.incrementarItem(umLivro,2)

 ESCREVA('incrementarItem - teste de falha 2: falhou')

CAPTURE Exception.livroÉNulo

 ESCREVA('incrementarItem - teste de falha 2: passou')

FIM TENTE

umCarrinho.destroy()

FIM MÉTODO

MÉTODO testeFalha3IncrementarItem()

-- item->select(livro=umLivro).quantidade+umaQuantidade >
umLivro.quantidadeEmEstoque

VAR FIXTURE umLivro:Livro

VAR FIXTURE umCarrinho:Carrinho

VAR FIXTURE umItem:Item

```
umLivro:=Livro.Create('0752201360', 'Bring me the head of Willy
    the mailboy!', 'Scott Adams', US$12,30, 128, 5)
-- a quantidade em estoque é 5.
umCarrinho:=Carrinho.Create()
-- o id do carrinho é definido internamente
umItem:=Item.Create(umLivro,umCarrinho,3)
TENTE
    umCarrinho.incrementarItem(umLivro,3)
    -- análise de valor limite
    ESCREVA('incrementarItem - teste de falha 3: falhou')
CAPTURE Exception.quantidadeAcimaDoEstoque
    ESCREVA('incrementarItem - teste de falha 3: passou)
FIM TENTE
umItem.destroy()
umCarrinho.destroy()
umLivro.destroy()
FIM MÉTODO
```

FIM CLASSE

O estilo do código para o caso de teste depende da linguagem e framework usados. Aqui, um pseudocódigo genérico foi ilustrado para permitir que a lógica do teste fosse compreendida por leitores com conhecimento em linguagens de programação, mas não necessariamente experiência com frameworks de teste.

Como foi visto no código anterior, *fixtures* e procedimentos de teste são usualmente similares e, assim, métodos uniformes e parametrizados poderiam ser usados para criar estes objetos, reduzindo a quantidade de código que seria produzida. Isso é usualmente disponibilizado por frameworks do tipo *xUnit*.

Testes de unidade podem ser extremamente minimizados se um gerador de código automatizado for usado para gerar as operações básicas e métodos delegados. Nesse caso, poder-se-ia assumir que o código está correto, porque o gerador não comete erros humanos. O teste poderia iniciar no nível de operações de sistema, conforme será explicado na próxima seção.

11.6 Teste de operações de sistema

O nível mais alto de teste ocorre quando a equipe verifica se as operações de sistema realmente são executadas de acordo com as especificações do contrato.

Operações de sistema podem invocar métodos de várias classes, delegar responsabilidades e coordenar a execução de métodos básicos e delegados. Assim, no nível de integração de classes que realizam um conjunto consistente de responsabilidades, o teste de operação de sistema verifica se cada operação de sistema implementa corretamente seu contrato.

O teste das operações de sistema, assim, consiste em verificar se, em condições normais (precondições verdadeiras e condições de exceção falsas), as pós-condições são realmente obtidas, e, se em condições anormais, as exceções são efetivamente ativadas. Testes de

operação de sistema são muito similares aos testes de unidade, exceto que agora a operação tem um contrato, e isso facilita muito a identificação dos casos de teste necessários.

Vamos examinar a operação de sistema *adicionarAoCarrinho* da Figura 10.13. Seu contrato foi definido no Capítulo 8 e é o seguinte:

```
Context Livir::adicionarAoCarrinho (umIdDeCarrinho:IdDeCarrinho,
    umISBN:ISBN, umaQuantidade:Natural)
def: umCarrinho=carrinho[umIdDeCarrinho]
def: umLivro=livro[umISBN]
def: umItem=umCarrinho.item->select(livro.isbn=umISBN)
pre:
    umCarrinho->notEmpty() and
    umLivro->notEmpty()
post:
    if umItem->isEmpty() then
        novoItem.isNewInstanceOf(Item) and
        umCarrinho^addItem(novoItem) and
        novoItem^setQuantidade(umaQuantidade) and
        novoItem^addLivro(umLivro) and
        novoItem^setPreçoUnitário(umLivro.preço)
    else
        umItem^setQuantidade(umItem.quantidade@pre+umaQuantidade)
    endif
exception:
    umItem.quantidade+umaQuantidade>umLivro.quantidadeEmEstoque
    implies Exception.throw('Quantidade não disponível')
```

A precondição *umCarrinho->notEmpty()* implica que há um conjunto inválido para o parâmetro *umIdDeCarrinho*, já a precondição *umLivro->notEmpty()* implica que há um conjunto inválido para o parâmetro *umISBN*. Finalmente, a exceção no contrato estabelece que há valores inválidos para *umaQuantidade* também.

Em relação a valores válidos, cada parâmetro deve ter pelo menos um conjunto de equivalência válido, caso contrário, o método sempre falharia. Entretanto, como as pós-condições no exemplo são condicionais (elas incluem uma estrutura *if-then-else-endif*), existem pelo menos dois comportamentos possíveis para essa operação, e isso define dois conjuntos de equivalência válidos: um no qual *umItem* é vazio e outro no qual *umItem* não é vazio. Os casos de teste para esta operação de sistema são então apresentados na Tabela 11.2.

Combinações de condições excepcionais poderiam também ser testadas. Por exemplo, uma situação que aparece na Tabela 11.2 consiste em ter tanto o ISBN quanto o *id* do carrinho inválidos. Se este fosse o caso, apenas uma das exceções poderia ser ativada de qualquer forma. Além disso, como mencionado anteriormente, algumas vezes, condições inválidas não são compatíveis. Por exemplo, se o ISBN for inválido, não há *quantidadeEmEstoque* para ser comparada à quantidade solicitada, e assim as exceções correspondentes nunca poderiam acontecer juntas.

Tabela 11.2 Casos de teste funcional para uma operação de sistema

Tipo	Condição a ser testada	Resultado esperado
Sucesso	umCarrinho->notEmpty() and umLivro->notEmpty() and umItem->isEmpty() and umaQuantidade <= umLivro.quantidadeEmEstoque	umCarrinho.item->select(livro=umLivro).quantidade = umaQuantidade
Sucesso	umCarrinho->notEmpty() and umLivro->notEmpty() and umItem->notEmpty() and umItem.quantidade + umaQuantidade <= umLivro.quantidadeEmEstoque	umItem.quantidade = umItem.quantidade@pre + umaQuantidade
Exceção	umCarrinho->isEmpty()	'Não existe este carrinho'
Exceção	umLivro->isEmpty()	'Não existe este livro'
Exceção	umItem->notEmpty() and umItem.quantidade + umaQuantidade > umLivro.quantidadeEmEstoque	'Quantidade não disponível'

É por isso que usualmente as únicas combinações que são testadas são aquelas que apenas incluem condições de sucesso ou apenas uma condição de falha combinada com condições de sucesso. Entretanto, se a equipe achar viável e mais seguro adicionar novas variações que incluam combinações de condições inválidas, elas também podem ser implementadas.

Assim, embora nem todas as combinações possam ser testadas, é necessário testar pelo menos:

- Todas as combinações possíveis de conjuntos válidos.
- Todas as combinações de um conjunto inválido com conjuntos válidos.

Isso significa que, após testar as combinações válidas, o testador deve incluir um conjunto inválido de cada vez para os testes de exceção.

O código para testar operações de sistema é muito similar ao apresentado para testes de unidade: *fixtures* devem ser criadas antes de cada teste, condições de sucesso e falha são testadas e o ambiente é finalmente limpo ao final de cada teste.

11.7 Teste de caso de uso (testes de sistema, aceitação e ciclo de negócio)

O teste de caso de uso consiste em manual ou automaticamente seguir todos os caminhos possíveis de um caso de uso e verificar se os resultados desejados (usualmente estabelecidos como pós-condições do caso de uso) são obtidos.

Se o teste é executado sobre um único caso de uso pela equipe de desenvolvimento, então ele é um *teste de sistema*. Se o teste é executado pelo cliente ou sob sua supervisão, então ele é um *teste de aceitação*. Se um conjunto completo de casos de uso de sistema relacionados a um caso de uso de negócio é executado em uma sequência lógica, como a definida pelo diagrama de atividade de negócio, então ele é um *teste de ciclo de negócio*.

O teste de caso de uso tem como objetivo verificar se a versão corrente do sistema permite executar corretamente os processos do caso de uso do ponto de vista de um usuário que realiza a sequência de operações de sistema em uma interface (não necessariamente gráfica) e se ele é capaz de obter os resultados esperados.

O teste de caso de uso pode ser entendido como a execução sistemática dos fluxos do caso de uso. Se cada uma das operações de sistema já passou pelos seus próprios testes, então deve ser verificado se o fluxo principal e os fluxos alternativos do caso de uso podem ser realizados corretamente e se eles produzem os resultados desejados.

É possível automatizar esses testes de forma que eles possam ser realizados por um robô de teste que faça chamadas diretamente ao controlador-fachada ou simulando as ações do usuário em uma interface gráfica.

Considere, por exemplo, um teste de sistema que deve ser conduzido para avaliar o caso de uso da Figura 5.7, reproduzido aqui como Figura 11.1.

Um teste de caso de uso deveria lidar com cenários que são de sucesso, mas executados por meio de diferentes caminhos. Primeiramente, o fluxo principal do caso de uso deve ser considerado um cenário base para propósito de teste. Então, cada variante ou exceção pode ser incluída, uma de cada vez, para testar cenários alternativos. Isso é o mínimo que é esperado de uma sequência de teste de caso de uso. Combinações de exceções e variantes podem ser testadas também se a equipe considerar que elas são úteis e viáveis.

A técnica para elaborar testes de caso de uso então consiste em selecionar um conjunto de caminhos que passe pelo menos uma vez por cada variante e fluxo de exceção. Os casos de teste para o exemplo da Figura 11.1 podem ser definidos como segue:

- 1, 2, 3, 4, 5.
- 1, 2, 3a(1, 2, 3), 4, 5.
- 1, 2, 3, 4, 5a(1,2), 5.
- 1, 2, 3, 4, 5b(1,2), 5.
- 1, 2, 3, 4, 5c(1), 1, 2, 3, 4, 5.
- 1, 2, 3, 4, 5d(1), 1, 2, 3, 4, 5.

O plano de teste para o caso de uso da Figura 11.1 é mostrado na Tabela 11.3.

Caso de uso 01: *Pedir livros*

1. O comprador fornece palavras-chave para pesquisar livros.
2. O sistema gera uma lista de livros para venda que satisfaz as palavras-chave incluindo pelo menos título, autor, preço, número de páginas, editora, ISBN e imagem de capa.
3. O comprador seleciona livros da lista e indica a quantidade desejada para cada um.
4. O sistema gera um resumo do pedido (título, autor, quantidade, preço unitário e subtotal para cada livro) e o valor total.
5. O comprador finaliza o pedido.

Exceção 3a: A quantidade solicitada é maior do que a quantidade em estoque para um ou mais livros.

- 3a.1. O sistema informa as quantidades disponíveis para cada livro que foi pedido acima do estoque.
- 3a.2. O comprador altera as quantidades desejadas para valores iguais ou menores do que o disponível no estoque.
- 3a.3. Avança para o passo 4.

Exceção 5a: O comprador ainda não se identificou.

- 5a.1. O comprador fornece uma identificação válida.
- 5a.2. Retorna ao passo 5.

Exceção 5b: O comprador não tem uma conta.

- 5b.1. O comprador se registra fornecendo nome de usuário, senha e endereço de email.
- 5b.2. Retorna ao passo 5.

Exceção 5c: Nenhum livro foi selecionado para compra.

- 5c.1. Retorna ao passo 1.

Variante 5d: O usuário deseja pesquisar mais livros.

- 5d.1. Retorna ao passo 1.
-

Figura 11.1

Caso de uso de referência.

Tabela 11.3 Exemplo de um plano de teste para um caso de uso.

Objetivo	Caminho	Como testar	Resultado
Fluxo principal	1, 2, 3, 4, 5	Um comprador previamente identificado pesquisa e seleciona pelo menos um livro e finaliza o pedido.	Um pedido é criado com o(s) livro(s) selecionado(s).
Exceção 3a	1, 2, 3a(1, 2, 3), 4, 5	Um comprador previamente identificado pesquisa e seleciona pelo menos um livro indicando quantidade que está acima do estoque. Então ele muda a quantidade para ficar consistente com o estoque e finaliza o pedido.	Um pedido é criado com o(s) livro(s) selecionado(s).
Exceção 5a	1, 2, 3, 4, 5a(1,2), 5	Um comprador registrado que ainda não foi identificado pesquisa e seleciona pelo menos um livro. Ele então fornece identificação e finaliza o pedido.	Um pedido é criado com o(s) livro(s) selecionado(s).
Exceção 5b	1, 2, 3, 4, 5b(1,2), 5	Um comprador não registrado pesquisa e seleciona pelo menos um livro. Ele se registra e finaliza o pedido.	Um pedido é criado com o(s) livro(s) selecionado(s). Um registro para um novo comprador é criado.
Exceção 5c	1, 2, 3, 4, 5c(1), 1, 2, 3, 4, 5	Um comprador previamente identificado tenta finalizar um pedido vazio e isso é evitado pelo sistema. Então ele pesquisa e seleciona pelo menos um livro e finaliza o pedido.	Um pedido é criado com o(s) livro(s) selecionado(s).
Variante 5d	1, 2, 3, 4, 5d(1), 1, 2, 3, 4, 5	Um comprador previamente identificado pesquisa e seleciona pelo menos um livro. Então ele pesquisa e seleciona outro livro e finaliza o pedido.	Um pedido é criado com os livros selecionados.

O teste de caso de uso somente é realizado sobre uma versão do sistema na qual todas as operações de sistema tenham sido já testadas. Se muitos erros ainda forem detectados nas operações durante o teste de sistema, a abordagem usual é abortar o teste de caso de uso e refazer os testes de operação de sistema até que uma versão suficientemente estável seja produzida e disponibilizada para os testes de casos de uso.

Embora o teste automático de casos de uso⁵ possa verificar se a funcionalidade está correta ou não, o teste de caso de uso executado por humanos é também recomendado porque há aspectos do teste que são difíceis para uma máquina avaliar. Por exemplo, as mensagens de erro são claras? Os botões e os campos de texto estão bem posicionados e dimensionados?

Além disso, um conjunto completo de *testes não funcionais* pode ser necessário dependendo das especificações suplementares e dos requisitos não funcionais do sistema (Capítulo 3). Exemplos de testes não funcionais são compatibilidade, conformidade, resistência, carga, localização, performance, recuperação, resiliência, segurança, escalabilidade, estresse e usabilidade. Explicar todos esses tipos de teste ultrapassa o escopo deste livro. Os testes não funcionais demandam técnicas heterogêneas e são cobertos por um significativo grupo de livros tais como: Molyneaux (2009) sobre teste de performance, Nelson (2004) sobre teste de estresse e Nielsen (1994) sobre teste de usabilidade.

⁵ Um exemplo de framework de robô para teste de interface está disponível em: <[HTTP://robotframework.googlecode.com/hg/doc/iserguide/RObotFrameworkUserGuide.html?#2.7.4](http://robotframework.googlecode.com/hg/doc/iserguide/RObotFrameworkUserGuide.html?#2.7.4)>. Acesso em: 1 set. 2013.

11.8 O processo visto aqui

	Concepção	Elaboração
Modelagem de negócio		
Requisitos		
Análise e design		
Implementação		
Teste		Planejar e executar o teste: • Realizar testes de unidade para métodos e classes que não foram gerados automaticamente. • Realizar testes de operação de sistema baseados em contratos de operação de sistema. • Realizar testes de caso de uso baseados em cenários de caso de uso.
Gerenciamento de Projeto		

11.9 Questões

1. Considere os métodos definidos na Seção 10.2 para a classe da Figura 10.1, e escreva uma sequência de testes de unidade para cada uma das operações básicas definidas para cada classe.
2. Observe os quatro contratos para *deletarLivro* definidos na Seção 8.7.3, e escreva um teste de sistema em sua linguagem favorita ou em pseudocódigo para testar cada uma das versões dessa operação.
3. Prepare uma tabela de teste de caso de uso para o caso de uso da Figura 5.4.

Modelagem da camada de interface com IFML

12

TÓPICOS-CHAVE NESTE CAPÍTULO

- Linguagem de Modelagem de Fluxo de Interação (IFML)
 - Componentes de visão
 - Organização de hipertexto
 - Padrões de páginas web
 - Operações em IFML
-

12.1 Introdução ao design da camada de interface

O design da camada de interface de um sistema depende fundamentalmente dos casos de uso detalhados ou diagramas de sequência de sistema, porque a interface deve permitir a um usuário seguir todos os fluxos de casos de uso. Diagramas de classe de design e contratos também podem ser úteis.

Durante o design da interface, os requisitos não funcionais que foram anotados nos casos de uso devem ser revisados novamente porque eles podem conter indicadores de como fazer o design da interface.

A linguagem de modelagem de fluxo de interação (Interaction Flow Modeling Language IFML) é uma extensão da UML para modelar interfaces com uso intensivo de dados. Ela é uma evolução de WebML (Ceri, Fraternali, Bongio, Brambilla, Comai & Matera, 2003). IFML está sendo adotada como um padrão pela OMG¹.

Este capítulo tem como objetivo apresentar os conceitos mais fundamentais de IFML e seu potencial de modelagem. Mais informação pode ser encontrada no *site* oficial da linguagem². Como IFML ainda está evoluindo como padrão, o leitor pode esperar encontrar algumas discrepâncias entre este texto e a especificação corrente de IFML, a qual pode ter mudado após a publicação deste livro.

12.2 Linguagem de Modelagem de Fluxo de Interação (IFML)

IFML é voltada para modelagem de interfaces, especialmente aquelas que fazem uso intensivo de dados, como sistemas de informação. Seu método de desenvolvimento propõe a construção de cinco modelos para definir uma aplicação:

¹ IFML é atualmente um projeto de padrão em andamento, e o primeiro rascunho foi publicado em março de 2013. Ele pode ser acessado em <<http://www.omg.org/spec/IFML>>.

² Disponível em: <www.ifml.org/>.

- *Modelo estrutural*: ele representa as estruturas e organização dos dados. Ceri *et al.* (2003) utilizam o diagrama entidade-relacionamento para definir o modelo estrutural. Entretanto, o diagrama de classes de UML (Capítulos 6 e 7) pode ser usado com os mesmos propósitos.
- *Modelo de derivação*: ele foi originalmente proposto para modelar dados que podem ser calculados a partir de outros dados. O modelo de derivação pode ser entendido como o conjunto de definições de atributos e associações derivadas que usualmente são definidos no modelo conceitual e podem ser escritos em OCL (Seções 6.2.3 e 6.4.3).
- *Modelo de composição*: ele inclui a definição de *contêineres* e agregados de *componentes de visão* básicos (Seção 12.3), *páginas* (Seção 12.4) e *áreas* (Seção 12.6.2).
- *Modelo de navegação*: onde os fluxos entre as páginas e componentes de visão são definidos (Seções 12.5 e 12.6).
- *Modelo de apresentação*: no qual o posicionamento dos componentes de visão da interface e sua aparência são definidos.

Este capítulo trata particularmente dos modelos de composição e navegação. Os modelos estrutural e de derivação já foram discutidos nos Capítulos 6 e 7. Para o modelo de apresentação, a ferramenta WebRatio³ permite o uso de planilhas de estilo XSL (*Extensible Stylesheet Language* – Linguagem Extensível de Folha de Estilos)⁴. A ferramenta de geração de código usa essas regras de apresentação para produzir páginas de acordo com o design produzido com outras ferramentas. Para os exemplos apresentados neste capítulo, as interfaces padrão renderizadas pela ferramenta, sem o uso de estilos, são mostradas. Mesmo os nomes dos campos foram deixados idênticos aos nomes dos atributos das classes com o objetivo de promover a compreensão mais do que a qualidade gráfica.

12.3 Componentes de visão

Componentes de visão são os elementos mais básicos em uma especificação IFML. Podem ser associados a classes de design e permitem a visualização de instâncias dessas classes.

Existem vários tipos de componentes de visão em IFML. Alguns exemplos são:

- *Detalhes*: mostra informação sobre um único objeto.
- *Detalhes múltiplos*: mostra informação sobre uma coleção de instâncias da mesma classe.
- *Lista simples*: mostra múltiplas instâncias de uma classe na forma de uma lista.
- *Lista*: um melhoramento da *lista simples* que permite *scrolling* e ordenação dinâmica.
- *Lista de múltipla escolha*: uma lista que permite seleção múltipla.
- *Hierarquia*: mostra múltiplas instâncias de diferentes classes organizadas em uma hierarquia, como mestre-detalle ou todo-parte.

³ A primeira ferramenta compatível com IFML.

⁴ Disponível em: <www.w3.org/Style/XSL/>.

- *Hierarquia recursiva*: mostra múltiplas instâncias de uma única classe organizadas hierarquicamente, como no caso de estruturas organizacionais.
- *Scroller*: permite as operações típicas de *scroll* sobre uma sequência de objetos.
- *Formulário*: permite entrada de dados baseada em formulário.
- *Formulário múltiplo*: similar ao formulário, mas permite a edição de múltiplos objetos de cada vez.
- *Mensagem*: permite imprimir mensagens em uma página.
- *Calendário*: mostra um calendário perpétuo.

Existem outros componentes de visão, e novos componentes podem ser definidos. Este capítulo apresenta informações sobre os mais fundamentais dentre eles. Para cada tipo de componente de visão, uma definição gráfica é apresentada, bem como a possível aparência da página renderizada por WebRatio. Todos os exemplos são baseados no DCD da Figura 12.1.

12.3.1 Detalhes

O componente de visão *detalhes* apresenta informação sobre um único objeto de uma dada classe. Ele é definido pelas seguintes propriedades:

- Um *nome* que é escolhido pelo designer.
- Uma *entidade* ou *fonte de dados* que é usualmente uma classe do diagrama de classes de design.
- *Atributos exibidos* que é uma lista de atributos da classe a serem exibidos quando a visão de detalhes é renderizada.

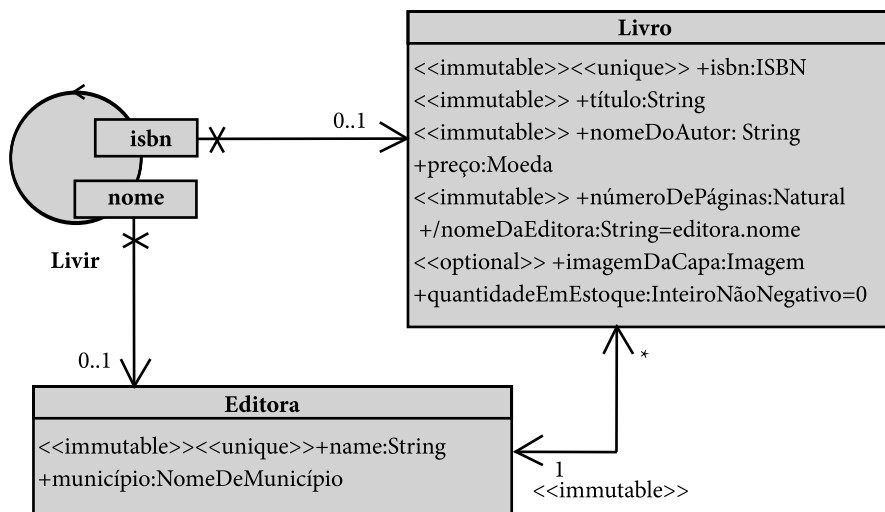


Figura 12.1

Diagrama de classes de design de referência.

A maioria dos componentes de visão como *detalhes* também permite a definição de uma ou mais *expressões condicionais* que funcionam como seletores ou filtros sobre o conjunto de instâncias a serem exibidas. Existem três tipos de expressões condicionais:

- *Condição-chave*: cada objeto no modelo de dados é identificado por um *OID* (*Object Identifier*) – um atributo que é único, obrigatório e imutável. Ele não deve ser confundido com nenhum dos atributos da classe conceitual, mesmo os estereotipados com <<unique>> e <<immutable>>, como ISBN ou CPF. O OID é um código gerado internamente e corresponde à *chave primária* de um objeto. *OIDs não podem ser conhecidos ou atualizados pelo usuário*. Uma condição-chave permite que sejam selecionadas uma ou mais instâncias de uma dada classe baseando-se no valor do seu atributo *OID*.
- *Condição de atributo*: ela permite a seleção de uma ou mais instâncias baseando-se nos valores de seus atributos. Operações como *maior que*, *igual* ou *contém* podem ser usadas para comparar o valor de um atributo com um dado parâmetro.
- *Condição de papel*: ela permite a seleção de uma ou mais instâncias baseando-se nas suas ligações.

A Figura 12.2 exibe um componente de visão *detalhes* que deve mostrar dados sobre uma instância de um livro para um dado ISBN. A expressão condicional *isbn=0517149257* determina qual instância é mostrada. Se esta instância não existisse, então nenhuma informação seria renderizada por aquele componente. Como o ISBN é um atributo que é único e obrigatório, é seguro assumir que não mais do que um livro é apresentado por esta visão de detalhes.

A Figura 12.3 mostra uma possível renderização web para o componente de visão de detalhes mostrado na Figura 12.2.

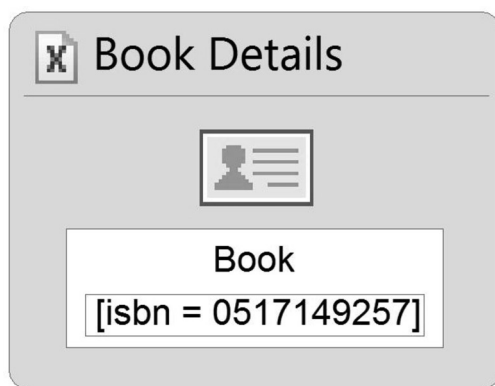


Figura 12.2

Um componente de visão de detalhes com uma condição de atributo simples.

Detalhes de Livro

título	The Ultimate Hitchhicker's Guide
nomeDoAutor	Douglas N. Adams
isbn	0517149257
preço	122
númeroDePáginas	815
imagemDaCapa	
quantidadeEmEstoque	4

Figura 12.3

Uma possível renderização para um componente de visão de detalhes com todos os atributos da classe *Livro*.

Detalhes de Livro

título	The Ultimate Hitchhicker's Guide
nomeDoAutor	Douglas N. Adams

Figura 12.4

Uma possível renderização de uma visão de detalhes com um número menor de atributos selecionados.

A renderização final da página pode variar, porque é possível adicionar definições de estilo de forma que interfaces com diferentes design e aparência possam ser produzidas.

Se o componente de visão de detalhes da Figura 12.2 for mudado para permitir que apenas os atributos *título* e *nomeDoAutor* apareçam, ela produziria o resultado renderizado na Figura 12.4.

12.3.2 Detalhes múltiplos

Detalhes múltiplos apresenta um grupo de instâncias de uma única classe de uma vez. Além das três propriedades já mostradas para o componente de visão de detalhes, ela adiciona as seguintes propriedades:

- *Atributos de ordenação*: são os atributos usados para ordenar as instâncias.
- *Máximo de resultados*: especifica o número máximo de instâncias que podem ser recuperadas. Por *default*, todas as instâncias são recuperadas.
- *Distinto*: especifica que elementos duplicados não serão mostrados mais do que uma vez.

O componente de *detalhes múltiplos* também permite a definição e o uso de expressões condicionais para selecionar quais elementos devem ser mostrados. A Figura 12.5 apresenta um exemplo de detalhes múltiplos.

**Figura 12.5**

Um componente de visão de detalhes múltiplos.

Detalhes Múltiplos de Livros						
isbn	título	nomeDoAutor	preço	número De Páginas	imagem Da Capa	quantidade Em Estoque
➤ 0671746723	Dirk Gently's Holistic Detective Agency	Douglas N. Adams	6.99	306		12
➤ 0671742515	The Long Dark Tea-Time of the Soul	Douglas N. Adams	6.99	307		4
➤ 0517149257	The Ultimate Hitchhicker's Guide	Douglas N. Adams	122	815		4

Figura 12.6

Uma possível renderização para um componente de visão de detalhes múltiplos.

**Figura 12.7**

Uma visão de detalhes múltiplos com uma expressão de condição composta.

A Figura 12.6 apresenta uma possível renderização para detalhes múltiplos da Figura 12.5 considerando que apenas três livros satisfazem a expressão condicional e que a ordenação acontece pelo título.

Em oposição ao componente de visão de detalhes que mostra apenas uma instância de cada vez, detalhes múltiplos apresentam todos os objetos que satisfazem a expressão condicional.

Se for necessário usar mais do que um atributo para determinar os objetos a serem apresentados pelo componente, então uma combinação de expressões condicionais pode ser usada. Essa combinação pode usar os operadores *and* e *or*. A Figura 12.7 apresenta um componente de visão de detalhes múltiplos que mostra instâncias que têm *númeroDePáginas* maior que 100 e *preço* menor do que R\$ 150,00.

12.3.3 Lista simples

A *lista simples* apresenta um ou mais atributos de um conjunto de instâncias de uma classe na forma de uma lista. Enquanto a *detalhes múltiplos* é normalmente usada para visualizar os detalhes de vários objetos em uma mesma página, a lista simples e suas variações são usadas quando uma lista ou menu são necessários para o usuário selecionar um ou mais elementos e realizar ações com eles. As propriedades das listas simples são as mesmas do componente de visão de detalhes múltiplos.

É possível, por exemplo, definir listas simples para selecionar livros com base em seus títulos. A Figura 12.8 apresenta o componente de visão para uma lista simples de títulos de livros e a Figura 12.9 apresenta sua possível renderização. A ausência de uma expressão condicional no componente de visão de lista simples implica que todas as instâncias de *Livro* serão mostradas.

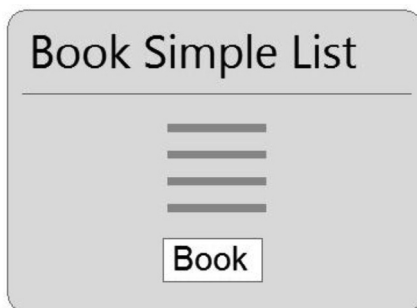


Figura 12.8

Lista Simples.

Book Simple List	
title	
> The Ultimate Hitchhicker's Guide	
> Dirk Gently's Holistic Detective Agency	
> Shave the Wales	
> The Revenge of the Baby-Sat	
> Foundation and Empire	
> Rama II	
> The Hammer of God	

Figura 12.9

Possível renderização para uma lista simples.

12.3.4 Lista

A *lista* é uma versão aprimorada da lista simples com operações de *scroll* e reordenação dinâmica dos elementos a partir de um simples clique no cabeçalho da respectiva coluna de atributo. Além das propriedades já definidas para listas simples, ela inclui as seguintes propriedades específicas:

- *Ordenável*: um atributo booleano que quando é verdadeiro define que a lista pode ser ordenada dinamicamente.
- *Atributos de ordenação default*: se a lista for ordenável, define os atributos de ordenação que podem ser selecionados pelo usuário para ordenar a lista e também o critério *default* de ordenação quando a lista é renderizada pela primeira vez.
- *Tamanho do histórico de ordenação*: especifica quantas ações de ordenação do usuário devem ser lembradas pela lista.
- *Assinalável*: se verdadeiro, uma *checkbox* é renderizada para cada elemento da lista.
- *Fator de bloco*: especifica quantos elementos são mostrados de cada vez. Se não foi especificado ou se definido como -1, todos os elementos são mostrados. Se o fator de bloco for definido, então os comandos de *scrolling* são renderizados para a lista.

A Figura 12.10 mostra o componente de visão IFML para uma lista. Esta lista é definida para a classe *Livro* com três atributos: *nomeDoAutor*, *título* e *preço*. O fator de bloco é definido como 3. A Figura 12.11 mostra uma versão renderizada desta lista.

Observe no topo da Figura 12.11 os comandos de *scroll* para a lista do primeiro para o último grupo de três elementos. Neste momento, os primeiros três elementos estão sendo mostrados, ordenados pelo seu título.

Se o usuário clicar em uma das ligações no topo de cada coluna, então os elementos da lista serão rearranjados e ordenados por aquela coluna. Se o usuário clicar, por exemplo, em *nomeDoAutor*, as linhas serão ordenadas pelos nomes dos autores, como mostrado na Figura 12.12.



Figura 12.10

Lista.

Lista de Livros			
firstprevious1 to 3 of 8nextlast jump to 1 2 3			
	<u>título</u>	<u>nomeDoAutor</u>	<u>preço</u>
➤	Dirk Gently's Holistic Detective Agency	Douglas N. Adams	6.99
➤	Foundation and Empire	Isaac Asimov	5.99
➤	Rama II	Arthur C. Clarke	6.99

Figura 12.11

Possível renderização de uma lista.

Lista de Livros		
firstprevious 1 to 3 of 8nextlast jump to 1 2 3		
<u>título</u>	<u>nomeDoAutor</u>	<u>preço</u>
➤ The Hammer of God	Arthur C. Clarke	6.99
➤ Rama II	Arthur C. Clarke	6.99
➤ The Revenge of the Baby-Sat	Bill Watterson	8.95

Figura 12.12

Renderização da Figura 12.11 após uma ação do usuário.

Lista de Seleção
Múltipla de Livros

☐ ☐ ☒ ☐
☐ ☐ ☐ ☐
☐ ☐ ☐ ☐
☐ ☐ ☐ ☐

Livro

Figura 12.13

Lista de seleção múltipla.

Note que os elementos mostrados são diferentes, porque a lista inteira é ordenada e os elementos que estão sendo vistos são os três primeiros.

12.3.5 Lista de seleção múltipla⁵

A *lista de seleção múltipla* permite que o usuário selecione um conjunto de elementos de uma lista. Fora isso, ela é muito similar a uma lista simples. A Figura 12.13 apresenta a definição IFML de uma lista múltipla, e a Figura 12.14 apresenta sua renderização.

⁵ Checkable list.

Enquanto as listas simples permitem a seleção de um único objeto de cada vez, as listas de seleção múltipla permitem a seleção de qualquer quantidade de objetos.

12.3.6 Formulários

Um *formulário* é usado para entrada de dados. Ele é muito útil para a criação de novas instâncias e também para fornecer parâmetros para pesquisas, consultas e comandos.

Um formulário é composto de um conjunto de *campos*. Cada campo contém um valor alfanumérico. Existem campos textuais, campos de seleção e campos de seleção múltipla.

Um formulário pode ser ou não associado a uma classe. Se ele for associado a uma classe, então seus campos podem ser associados aos atributos da classe.

Formulários que são usados para coletar parâmetros para uma consulta ou comando usualmente não precisam ser associados a classes. Por exemplo, um formulário que aceita palavras-chave para pesquisar livros não precisa estar associado a qualquer classe.

Os campos de um formulário têm as seguintes propriedades (além de outras):

- *Nome*: o nome do campo atribuído pelo designer.
- *Atributo*: se o formulário é associado a uma classe, então o campo pode ser associado a um de seus atributos.
- *Tipo de conteúdo*: define o tipo do campo, que pode ser *string*, *text*, *blob* ou *url*.

Lista de Seleção Múltipla de Livros

	título	nomeDoAutor	preço
☐	Shave the Wales	Scott Adams	6.99
☒	The Revenge of the Baby-Sat	Bill Watterson	8.95
☐	Foundation and Empire	Isaac Asimov	5.99
☒	Rama II	Arthur C. Clarke	6.99
☒	The Hammer of God	Arthur C. Clarke	6.99
☐	The Long Dark Tea-Time of the Soul	Douglas N. Adams	6.99
☐	Dirk Gently's Holistic Detective Agency	Douglas N. Adams	6.99
☐	The Ultimate Hitchhicker's Guide	Douglas N. Adams	122

Figura 12.14

Renderização para uma lista de seleção múltipla.

**Figura 12.15**

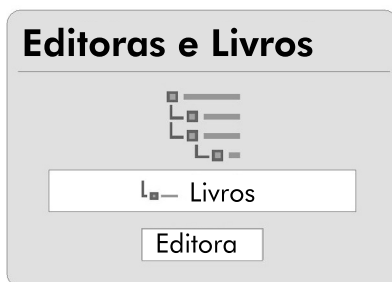
Um formulário.

Formulário de Livro

isbn	
título	
nomeDoAutor	
preço	
númeroDePáginas	
imagemDaCapa	
quantidadeEmEstoque	

Figura 12.16

Renderização de um formulário.


Figura 12.17

Uma hierarquia.

- *Pré-carregado*: o campo pode ser pré-carregado com valores que são mostrados ao usuário quando o formulário é renderizado.
- *Modificável*: define se o valor inicial de um campo pode ser trocado ou não. O *default* é que o campo pode ser atualizado.

A Figura 12.15 apresenta um formulário que é associado a classe *Livro* e a Figura 12.16 apresenta sua renderização.

12.3.7 Hierarquias

Hierarquias são usadas para mostrar dependências entre objetos com dois ou mais níveis. Elas são úteis, por exemplo, para mostrar objetos que se relacionam por uma associação de um para muitos, como *Editora* e *Livro*, ou parte-todo, como *Livro* e *Capítulo*. A Figura 12.17 mostra uma definição para uma hierarquia de dois níveis com *Editora* no topo e *Livro* no segundo nível.

O segundo nível usualmente define uma *expressão condicional baseada em papel*. No exemplo, a expressão condicional baseada em papel define que apenas livros que estiverem ligados a uma dada editora são mostrados abaixo dela. Na ausência dessa expressão condicional, todos os livros seriam mostrados abaixo de cada editora, mas agora este não é o caso.

A Figura 12.18 mostra uma renderização para a hierarquia da Figura 12.17 com livros ligados às suas respectivas editoras.

Existem situações que requerem a aplicação do padrão *hierarquia organizacional*, conforme explicado na Seção 7.8, onde um conceito tem subcomponentes que pertencem à mesma classe, a qual pode ser recursivamente aninhada. Para esses casos, IFML fornece o componente de visão chamado *hierarquia recursiva*. A Figura 12.19 mostra uma hierarquia recursiva definida para a classe *EstruturaOrganizacional* mostrada na Figura 7.15.

Editoras e Livros

- **nome** Bantam
 - **título** Dirk Getly's Holistic Detective Agency
 - **título** The Hammer of God
 - **título** Foundation and Empire
- **nome** Pocket Books
 - **título** The Long Dark Tea-Time of the Soul
 - **título** Rama II
- **nome** Boxtree
 - **título** Shave the Whales
- **nome** Andrews and McMeel
 - **título** The Revenge of the Baby-Sat
- **nome** Random House
 - **título** The Ultimate Hitchhicker's Guide

Figura 12.18

Renderização de uma hierarquia.

Estrutura Organizacional Recursiva



superestrutura

Figura 12.19

Uma hierarquia recursiva.

A definição e a renderização de uma hierarquia recursiva são similares à definição e renderização de uma hierarquia normal. As únicas diferenças são que a classe detalhe e a classe mestre são as mesmas e o número de níveis é indeterminado.

12.4 Páginas

Páginas são os elementos de interface que são efetivamente acessados pelo usuário. Tipicamente, uma página contém um ou mais componentes de visão e possivelmente outras páginas.

A representação IFML de uma página contém um nome e um conjunto opcional de componentes de visão ou subpáginas incluídos. Por exemplo, a Figura 12.20 apresenta uma página que contém uma lista de editoras e uma lista de livros e a Figura 12.21 apresenta sua renderização.

O símbolo  na Figura 12.20 indica que esta página é uma *homepage*, ou seja, é a página inicial de uma aplicação. O outro símbolo no canto superior direito da página não tem nenhum significado especial em IFML; trata-se apenas de um comando específico da ferramenta para destacar ou obscurecer elementos gráficos.

Os componentes de visão que aparecem dentro da página na Figura 12.20 são independentes um do outro porque não há fluxos (ver Seção 12.5) entre eles.

Por *default*, componentes de visão definidos na mesma página são renderizados um abaixo do outro e ocupam a largura total da página, mas é possível usar o *modelo de apresentação*, como mencionado anteriormente, para definir componentes de visão em diferentes posições e tamanhos. O modelo de apresentação consiste basicamente em uma grade na qual os componentes de visão são posicionados e dimensionados dentro de uma página. No exemplo da Figura 12.22, os componentes de visão são colocados lado a lado.



Figura 12.20

Uma página com dois componentes de visão.

12.5 Fluxos

Um *fluxo* é uma conexão orientada entre duas páginas ou componentes de visão. Fluxos podem ser usados não apenas para definir possibilidades de navegação entre páginas, mas também para definir dependências de dados. Por exemplo, selecionar um elemento em um componente de visão de lista pode fazer com que informação sobre o elemento apareça em um componente de visão de detalhes. Isso pode ser conseguido pela definição de um fluxo indo da lista para o componente de detalhes.

> Principal

Lista de Editoras					
nome	município				
> Andrews and McMeel	Kansas City				
> Bantam	New York				
> Boxtree	London				
> Pocket Books	New York				
> Random House	New York				

Lista de Livros					
isbn	título	nomeDoAutor	preço	número De Páginas	quantidade Em Estoque
> 0671746723	Dirk Getty's Holistic Detective Agency	Douglas N. Adams	6.99	306	12
> 0553293370	Foundation and Empire	Isaac Asimov	5.99	282	3
> 0553286587	Rama II	Arthur C. Clarke and Gentry Lee	6.99	466	2
> 0752208497	Shave the Whales	Scott Adams	6.99	128	7
> 055356871X	The Hammer of God	Arthur C. Clarke	6.99	240	1
> 0671742515	The Long Dark Tea-Time of the Soul	Douglas N. Adams	6.99	307	21
> 0836218663	The Revenge of the Baby-Sat	Bill Waterson	8.95	127	4
> 0517149257	The Ultimate Hitchhicker's Guide	Douglas N. Adams	129	813	6

Figura 12.21

Renderização de uma página com dois componentes de visão.

Lista de Editoras		Lista de Livros				
nome	município	isbn	título	nomeDoAutor	preço	número quantidade De Páginas Em Estoque
> Andrews and McMeel	Kansas City	> 0671746723	Dirk Getty's Holistic Detective Agency	Douglas N. Adams	6.99	306 12
> Bantam	New York	> 0553293370	Foundation and Empire	Isaac Asimov	5.99	282 3
> Boxtree	London	> 0553286587	Rama II	Arthur C. Clarke and Gentry Lee	6.99	466 2
> Pocket Books	New York	> 0752208497	Shave the Whales	Scott Adams	6.99	128 7
> Random House	New York	> 055356871X	The Hammer of God	Arthur C. Clarke	6.99	240 1
		> 0671742515	The Long Dark Tea-Time of the Soul	Douglas N. Adams	6.99	307 21
		> 0836218663	The Revenge of the Baby-Sat	Bill Waterson	8.95	127 4
		> 0517149257	The Ultimate Hitchhicker's Guide	Douglas N. Adams	129	813 6

Figura 12.22

Um arranjo de renderização diferenciado para a página mostrada na Figura 12.21.

12.5.1 Fluxo normal de navegação

Fluxos normais de navegação são um tipo de fluxo que é renderizado como uma ligação clicável em sua página ou componente de visão de origem. Quando uma ligação é clicada, ela provoca a navegação para a página ou componente de visão. Ela é representada por uma seta.

A Figura 12.23 mostra um exemplo de um fluxo de navegação normal entre duas páginas. A renderização resultante da página *Editoras* contém uma ligação para a página *Livros* (no canto superior direito da Figura 12.24). Podemos ver na Figura 12.23 que o fluxo não é ligado aos componentes de visão, mas às páginas. Portanto, a ligação será renderizada apenas uma vez na página na qual a lista de livros é visualizada.

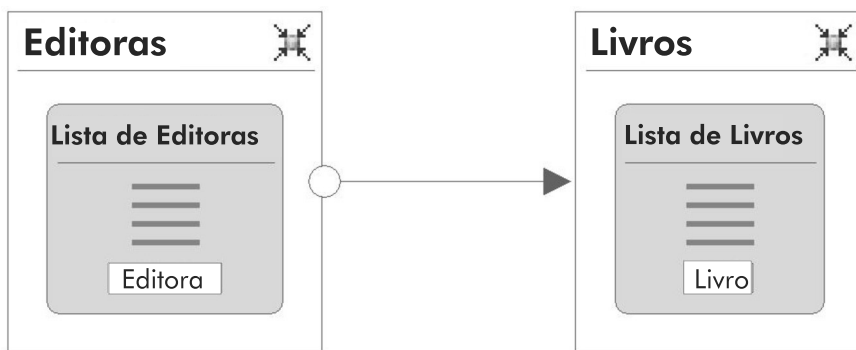


Figura 12.23

Um fluxo de navegação normal entre páginas.

[Livros](#)

> Editoras

Lista de Editoras

nome	município
> Bantam	New York
> Pocket Books	New York
> Boxtree	London
> Andrews and McMeel	Kansas City
> Random House	New York

Figura 12.24

Renderização de um fluxo de navegação normal como uma ligação.

Se o fluxo tem sua origem no componente de visão de lista, como na Figura 12.25, então cada linha na lista deve ter uma ligação individual para a página *Livros*.

A Figura 12.26 mostra como o modelo da Figura 12.25 poderia ser renderizado.

Esta lista é ainda apenas um conjunto de ligações que tem o mesmo efeito: navegar para a página *Livros*. As seguintes seções mostram como este tipo de fluxo pode ser usado para transportar informação de um componente de visão para outro, tornando possível, por exemplo, visualizar apenas os livros de uma dada editora.

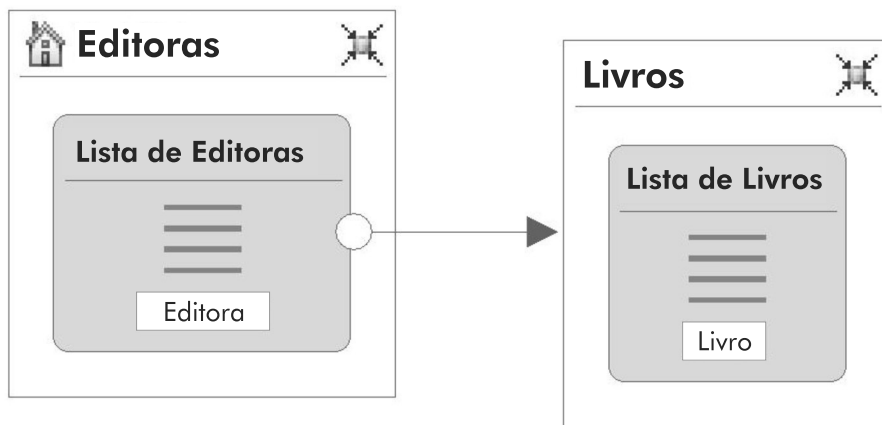


Figura 12.25

Um fluxo de um componente de visão para uma página.

> Editoras

Lista de Editoras

nome	município	
> Bantam	New York	Livros
> Pocket Books	New York	Livros
> Boxtree	London	Livros
> Andrews and McMeel	Kansas City	Livros
> Random House	New York	Livros

Figura 12.26

Renderização de uma ligação para cada elemento de uma lista.

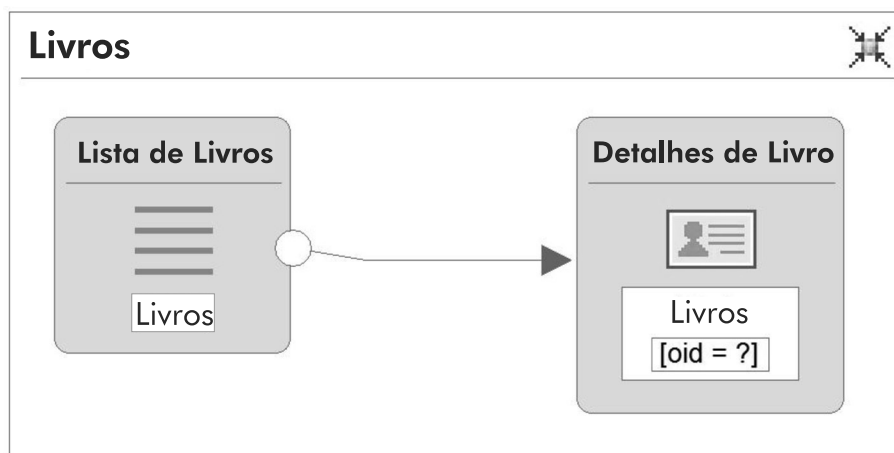


Figura 12.27

Um fluxo de navegação normal conectando dois componentes de visão.

12.5.2 Fluxo de dados

Fluxos de dados são um tipo de fluxo que não é renderizado, mas que define dependências entre componentes de visão⁶. Setas tracejadas representam fluxos de dados.

Fluxos de dados não definem navegação entre componentes de visão. Eles são usados para obter valores de um componente sem estar navegando a partir dele. Eles são particularmente úteis quando um componente necessita de dados de mais do que outro componente. As seções seguintes apresentam exemplos de fluxos de dados que obtêm dados de diferentes fontes.

12.5.3 Vínculo de parâmetro

Um *vínculo de parâmetro* é uma peça de informação que é transmitida da origem para o destino de um fluxo. Um vínculo de parâmetro tem um *nome* e um *valor*, o qual é obtido no componente de origem.

Um fluxo pode ter múltiplos vínculos de parâmetro, passando um conjunto de valores em vez de um único.

Usualmente, chaves ou valores de atributos de objetos são passados por vínculos de parâmetros. Por *default*, um fluxo de um componente de visão vincula a chave primária do objeto selecionado no componente de origem e a envia ao destino, onde ela pode ser usada, por exemplo, por uma expressão condicional.

Nos exemplos anteriores, apenas constantes foram usadas em expressões condicionais para selecionar os elementos a serem mostrados nos componentes de visão, como listas e

⁶ E operações, como explicado adiante.

detalhes. Agora, com fluxos e vínculos de parâmetro, é possível usar um componente de visão para definir quais elementos são mostrados em outro componente de visão.

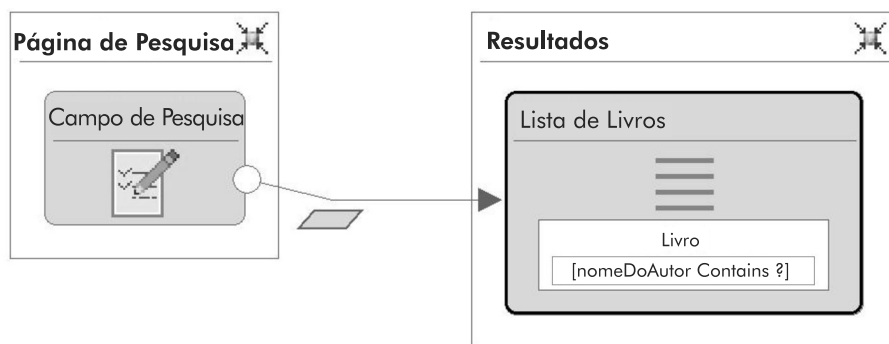
O diagrama IFML da Figura 12.27 mostra uma lista de livros com um fluxo para um componente de visão de detalhes. A visão de detalhes tem uma condição-chave [*oid=?*]. Como a chave para o livro selecionado na lista é vinculado ao fluxo que vai para a visão de detalhes, apenas aquele livro será mostrado quando a visão de detalhes for renderizada.

Quando o usuário seleciona um livro na lista, ao clicar na respectiva ligação, a visão de detalhes apresenta informações sobre o livro. A Figura 12.28 mostra o estado da página quando a ligação associada ao primeiro livro é clicada.

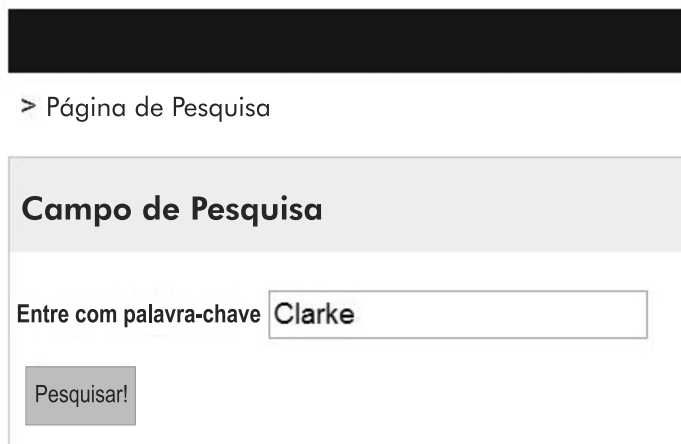
> Livros			
Lista de Livros			
	título	nomeDoAutor	
>	Foundation and Empire	Isaac Asimov	Detalhes
>	Rama II	Arthur C. Clarke and Gentry Lee	Detalhes
>	Shave the Whales	Scott Adams	Detalhes
>	The Hammer of God	Arthur C. Clarke	Detalhes
>	The Long Dark Tea-Time of the Soul	Douglas N. Adams	Detalhes
>	The Revenge of the Baby-Sat	Bill Waterson	Detalhes
>	The Ultimate Hitchhicker's Guide	Douglas N. Adams	Detalhes
Detalhes de Livros			
isbn	0553293370		
título	Foundation and Empire		
nomeDoAutor	Isaac Asimov		
preço	5.99		
númeroDePáginas	282		
imagemDaCapa			
quantidadeEmEstoque	7		

Figura 12.28

Renderização do modelo da Figura 12.27 quando o primeiro livro da lista é selecionado.


Figura 12.29

Um fluxo que transporta dados de um formulário para uma lista.



A renderização visual da interface mostra uma barra preta no topo. Abaixo dela, há um link '> Página de Pesquisa'. O formulário principal, intitulado 'Campo de Pesquisa', contém o texto 'Entre com palavra-chave' seguido por um campo de entrada com o valor 'Clarke'. Abaixo do campo, há um botão 'Pesquisar!'.

Figura 12.30

Renderização do modelo da Figura 12.29 antes de iniciar a pesquisa.

O paralelogramo que está próximo ao fluxo indica que este fluxo possui um vínculo de parâmetro que não é a chave de objeto *default*.

Na renderização mostrada na Figura 12.30, o usuário digitou “Clarke” no campo de pesquisa e os resultados produzidos são mostrados na Figura 12.31.

O vínculo de parâmetro também pode ser usado para associar um formulário a outro componente de visão. Por exemplo, o formulário pode fornecer um campo de pesquisa e uma lista poderia apresentar apenas os elementos que satisfazem o critério de seleção.

A Figura 12.29 apresenta um exemplo no qual um campo de pesquisa é usado para encontrar os livros de um dado autor. O usuário pode digitar o nome do autor ou parte dele no campo de pesquisa e os resultados são mostrados na lista *Livros*.

12.5.4 Vínculo de parâmetro multivalorado

É possível definir fluxos a partir de componentes de escolha múltipla, como listas múltiplas. Nesses casos, um vínculo de parâmetro passa um conjunto de valores (as chaves dos elementos selecionados, por exemplo).

Se uma lista múltipla é ligada a uma visão de detalhes múltiplos como na Figura 12.32, o usuário pode selecionar vários elementos da lista múltipla como na Figura 12.33 e pressionar o botão abaixo para navegar para a página que mostra os detalhes apenas dos livros selecionados (Figura 12.34). A visão de detalhes múltiplos tem uma condição-chave que estabelece que apenas livros cujo OID pertença ao conjunto que é vinculado ao fluxo são mostrados.

> Resultados

Lista de Livros						
isbn	título	nomeDoAutor	preço	número De Páginas	imagem Da Capa	quantidade Em Estoque
> 0553286587	Rama II	Arthur C. Clarke and Gentry Lee	6.99	466		12
> 055356871X	The Hammer of God	Arthur C. Clarke	6.99	240		2

Figura 12.31

Renderização do modelo da Figura 12.29 após a pesquisa ser realizada.



Figura 12.32

Um fluxo para vinculação de parâmetro multivalorada.

> Seleção

Selecionar livros para ver detalhes

titulo	nomeDoAutor
☐ Foundation and Empire	Isaac Asimov
☒ Rama II	Arthur C. Clarke and Gentry Lee
☐ The Hammer of God	Arthur C. Clarke
☐ The Long Dark Tea-Time of the Soul	Douglas N. Adams
☒ Shave the Whales	Scott Adams
☒ The Revenge of the Baby-Sat	Bill Waterson
☐ The Ultimate Hitchhicker's Guide	Douglas N. Adams

Figura 12.33

Renderização do modelo da Figura 12.32 antes de clicar o botão.

> Detalhes

Detalhes Livros						
isbn	titulo	nomeDoAutor	preço	número DePáginas	imagem DaCapa	quantidade EmEstoque
> 0553286587	Rama II	Arthur C. Clarke and Gentry Lee	6.99	466		12
> 0752208497	Shave the Whales	Scott Adams	6.99	128		0
> 0836218663	The Revenge of the Baby-Sat	Bill Waterson	8.95	127		21

Figura 12.34

Renderização do modelo da Figura 12.32 após clicar o botão.

12.6 Organização de hipertexto

Componentes de visão, fluxos e sua organização dentro das páginas constituem os modelos de interface mais locais. Entretanto, é também necessário criar modelos mais amplos para uma interface, incluindo grupos de páginas inter-relacionadas, regiões de navegação, *landmarks*, e assim por diante. Este tipo de modelo é descrito nesta seção e referenciado aqui como *organização de hipertexto*.

12.6.1 Visões de site

Uma *visão de site* é um pacote que contém uma aplicação inteira que pode ser disponibilizada para usuários. Ela usualmente contém um grupo de áreas (veja a seção seguinte) e *páginas* para a aplicação. A Figura 12.35 apresenta um exemplo de uma visão de site com uma página (*Principal*) e duas áreas (*Administração* e *Cliente*).

Este exemplo ainda não foi totalmente desenvolvido, porém. Apenas a estrutura de nível mais alto é mostrada na Figura 12.35.

12.6.2 Áreas

Uma visão de site pode ser organizada em áreas. Áreas e páginas são referenciadas em IFML como *contêineres*. Áreas são grupos de contêineres que têm afinidades, como, por exemplo, um conjunto de funcionalidades relacionadas a um objetivo de usuário.

No modelo IFML da Figura 12.35, as áreas aparecem como retângulos sombreados. As áreas oferecem organização hierárquica para uma visão de site.

12.6.3 Home, landmark e default

Na Figura 12.35, alguns contêineres têm propriedades especiais:

- [H] ou  indica a *homepage*. A homepage é a página inicial de uma aplicação. Quando um usuário acessar o web site, ele vai acessar a homepage por *default*. Áreas não podem ter essa propriedade.
- [D] indica a página *default* de uma área. Quando um usuário navega para uma área específica, ele vai iniciar na página *default*. Assim, a página *default* é uma espécie de “homepage” para uma área específica. Áreas não podem ter essa propriedade.
- [L] representa uma área ou página *landmark*. Isso significa que a página ou área é diretamente acessível de qualquer outra página na mesma área. Assim, isso evita a necessidade de criar muitos fluxos para páginas que devem ser acessadas de vários lugares.



Figura 12.35

Uma visão de site.



Figura 12.36

Exemplo de um índice cascata.

Apenas uma *homepage* pode existir para uma dada visão de site e apenas uma página *default* para uma dada área. Entretanto, qualquer número de páginas ou áreas *landmark* podem existir em qualquer contêiner.

12.7 Padrões de interface web

A maioria dos padrões de interface apresentados nesta seção é adaptada de Ceri *et al.* (2003) e Tidwell (2005).

12.7.1 Índice em cascata

Um índice em cascata é uma sequência de menus baseados em listas que no final atingem uma visão de detalhes. Por exemplo, um usuário pode iniciar selecionando uma editora; então ele pode selecionar um livro a partir de uma lista da editora e, finalmente, acessar os detalhes de um livro. A Figura 12.36 ilustra este padrão.

No diagrama da Figura 12.36, como os vínculos de parâmetro são os *default*, os fluxos carregam os OIDs dos elementos selecionados. Estes OIDs podem ser usados pelas expressões condicionais no componente de visão de destino para definir quais elementos devem ser mostrados.

O componente de visão *Lista de Livros* usa uma expressão condicional baseada em papel que mostra apenas os livros ligados à editora cujo OID é vinculado ao fluxo de entrada que vem de *Lista de Editoras*.

Note que *Lista de Livros* tem uma expressão condicional que é baseada no papel *EditoraParaLivro*, ou seja, apenas livros ligados à editora na origem do fluxo são mostrados por *Lista de Livros*.

A visão de detalhes *Livro* mostra a informação completa sobre o livro selecionado em *Lista de Livros*. Os OIDs dos livros selecionados são vinculados ao fluxo que vem de *Lista de Livros* para a visão de detalhes. As Figuras 12.37 a 12.39 mostram uma sequência de interfaces renderizadas que ilustram este padrão. Na Figura 12.37, uma editora pode

ser selecionada. Na Figura 12.38, após selecionar a segunda editora, um de seus livros pode ser selecionado. Finalmente, na Figura 12.39, o terceiro livro foi selecionado e seus detalhes são mostrados a seguir.

> Selecionar Editora

Lista de Editoras		
	nome	
>	Andrews and McMeel	Livros
>	Bantam	Livros
>	Boxtree	Livros
>	Pocket Books	Livros
>	Random House	Livros

Figura 12.37

Renderização de *Lista de Editoras*.

> Selecionar Livro

Lista de Livros		
	título	nomeDoAutor
>	Dirk Getly's Holistic Detective Agency	Douglas N. Adams Detalhes
>	Foundation and Empire	Isaac Asimov Detalhes
>	The Hammer of God	Arthur C. Clarke Detalhes

Figura 12.38

Renderização de *Lista de Livros*.

> Detalhes

Livro

isbn

055356871X

título

The Hammer of God

nomeDoAutor

Arthur C. Clarke

preço

6.99

númeroDePáginas

240

imagemDaCapa



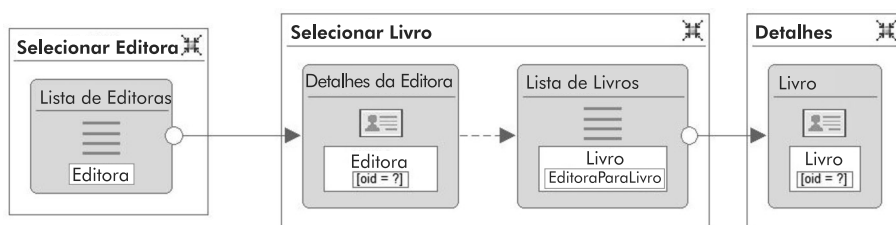
quantidadeEmEstoque

1

Figura 12.39

Renderização do componente de detalhes *Livro*.

Uma variação deste padrão consiste em mostrar alguns dados do objeto selecionado em uma das listas intermediárias. A Figura 12.40 ilustra essa situação.


Figura 12.40

Índice cascata com apresentação de dados intermediários.

Nesta figura, quando uma editora é selecionada, a interface mostra os detalhes da editora e uma lista de livros para seleção. Note que na página intermediária *Detalhes da Editora* está conectado a *Lista de Livros* por um fluxo de dados. Não é necessário que o usuário navegue de *Detalhes da Editora* para sua lista de livros: ambos os componentes de visão são mostrados ao mesmo tempo quando a página *Selecionar Livro* é acessada. A Figura 12.41 mostra como a página intermediária seria renderizada.

> Selecionar Livro

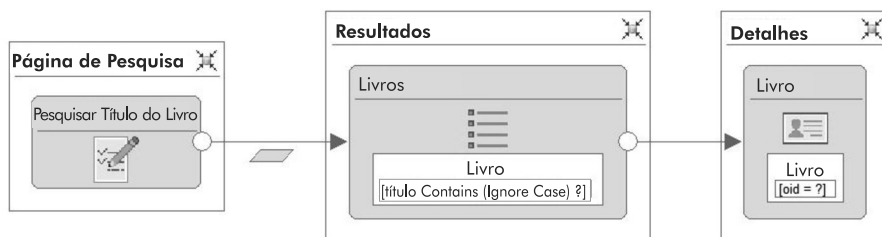
Detalhes da Editora
nome Bantam
município New York

Lista de livros

título	nomeDoAutor	
> Dirk Getly's Holistic Detective Agency	Douglas N. Adams	Detalhes
> Foundation and Empire	Isaac Asimov	Detalhes
> The Hammer of God	Arthur C. Clarke	Detalhes

Figura 12.41

Renderização da página intermediária de um índice cascata com dados intermediários.

**Figura 12.42**

Um exemplo de um índice filtrado.

12.7.2 Índice filtrado

Em alguns casos, pode ser necessário mostrar uma lista com elementos selecionados. Por exemplo, se a livraria tiver milhares de livros em seu catálogo, não seria viável para um usuário simplesmente procurar por um livro em uma lista. Uma lista filtrada poderia ser usada no lugar disso, por exemplo, pedindo ao usuário para fornecer uma palavra-chave de forma a apresentar uma lista de livros reduzida.

Este padrão é realizado por uma conexão entre um formulário, uma lista e um componente de detalhes, como mostrado na Figura 12.42.

Como os resultados são mostrados em uma lista (não uma lista *simples*), um fator de bloco de *scroll* pode ser definido, bem como a possibilidade de dinamicamente ordenar os resultados. Além disso, o número máximo de resultados pode ser definido para esta lista de forma que listas desnecessariamente grandes não são apresentadas.

12.7.3 Tour guiado

Um *tour guiado* é um padrão que permite que se visualize os detalhes de um conjunto de objetos, um de cada vez, usando operações de *scroll*. Ele é implementado pelo uso do componente *scroller* com *fator de bloco* 1 ligado a uma visão de detalhes, conforme mostra a Figura 12.43:

A Figura 12.44 mostra uma renderização desta especificação na qual o quinto de oito livros foi selecionado pelo usuário.

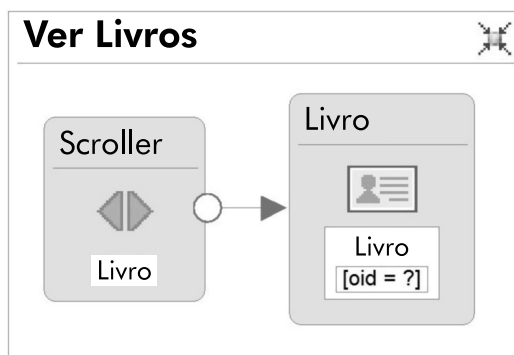


Figura 12.43

Especificação de um tour guiado.

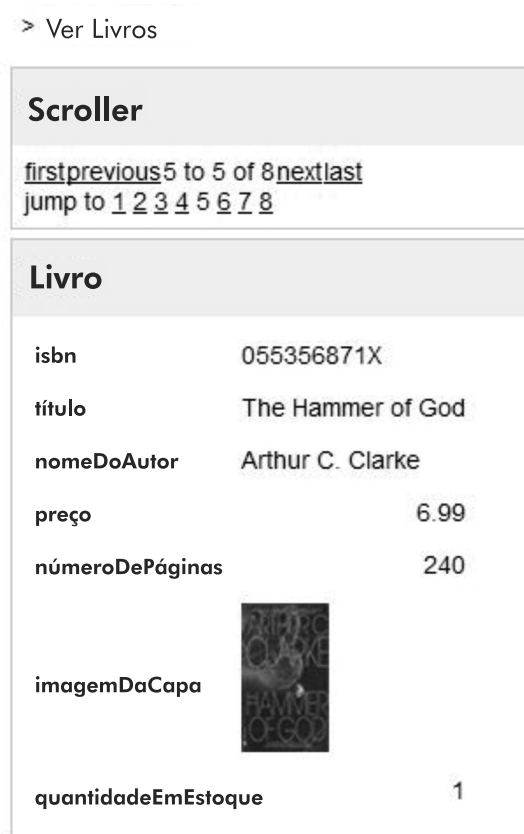


Figura 12.44

Renderização de um tour guiado.

12.7.4 Pontos de vista

Algumas vezes, pode ser interessante apresentar diferentes faces de certos objetos em diferentes momentos. Por exemplo, dados resumidos sobre um livro podem ser apresentados por *default*, mas esses dados podem ser expandidos ou contraídos novamente se o usuário desejar.

Para implementar o padrão de pontos de vista, dois ou mais componentes de visão de detalhes podem ser definidos para a mesma classe, com diferentes conjuntos de atributos. Fluxos podem ser definidos entre os diferentes componentes para permitir ao usuário mudar de uma visão para outra, como mostra a Figura 12.45:

A Figura 12.46 mostra a aparência do ponto de vista resumido, e a Figura 12.47 mostra o ponto de vista da informação completa.

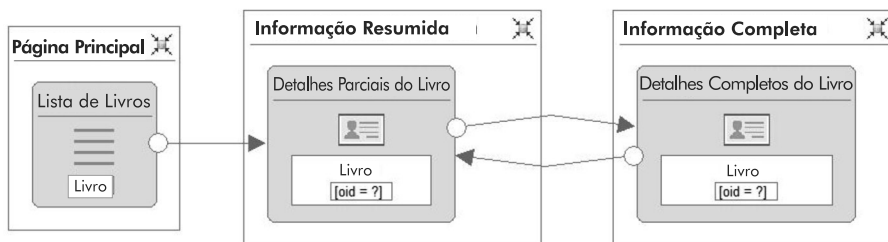


Figura 12.45

Especificação de pontos de vista.

> Informação Resumida

Detalhes Parciais do Livro

título	Dirk Getly's Holistic Detective Agency
nomeDoAutor	Douglas N. Adams
preço	6.99
mais detalhes	

Figura 12.46

Ponto de vista resumido.

12.7.5 Visão geral mais detalhe

O padrão *Visão geral mais detalhe* é muito comum em aplicações, como visualizadores de e-mail, mas ele pode ser aplicado a uma variedade de outras situações. A ideia é ter na mesma página uma lista com *scroll* de elementos e uma visão de detalhes de um elemento selecionado em uma região próxima da lista.

> Informação Completa

Detalhes Completos do Livro

isbn	0671746723
título	Dirk Getly's Holistic Detective Agency
nomeDoAutor	Douglas N. Adams
preço	6.99
númeroDePáginas	306
imagemDaCapa	
quantidadeEmEstoque	12

[menos detalhes](#)

Figura 12.47

Ponto de vista com informação completa.

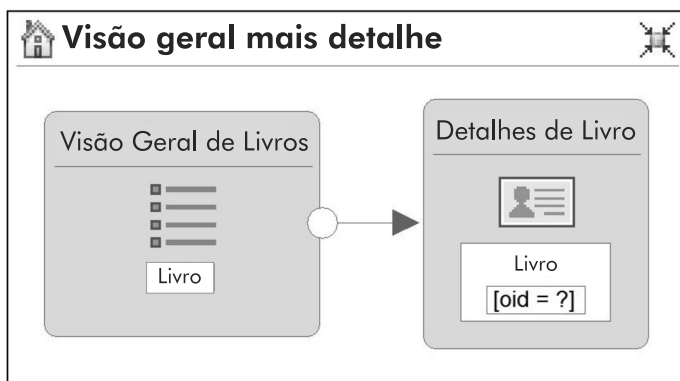


Figura 12.48

Especificação do padrão *visão geral mais detalhe*.

A Figura 12.48 mostra uma definição IFML que usa este padrão com uma lista com fator de bloco 3 e um componente de visão de detalhes ligado a ela por um fluxo.

A Figura 12.49 mostra uma renderização para a página especificada na Figura 12.48.

➤ Visão geral mais detalhe

Visão Geral de Livros

[first](#)[previous](#) 4 to 6 of 8 [next](#)[last](#)
[jump to 1](#) [2](#) [3](#)

título	nomeDoAutor	preço	
➤ The Long Dark Tea-Time of the Soul	Douglas N. Adams	6.99	mostrar detalhes abaixo
➤ Rama II	Arthur C. Clarke and Gentry Lee	6.99	mostrar detalhes abaixo
➤ The Hammer of God	Arthur C. Clarke	6.99	mostrar detalhes abaixo

Detalhes de Livro

isbn 0553286587

título Rama II

nomeDoAutor Arthur C. Clarke and Gentry Lee

preço 6.99

númeroDePáginas 466



quantidadeEmEstoque 2

Figura 12.49

Exemplo de renderização de uma página usando o padrão *visão geral mais detalhe*.

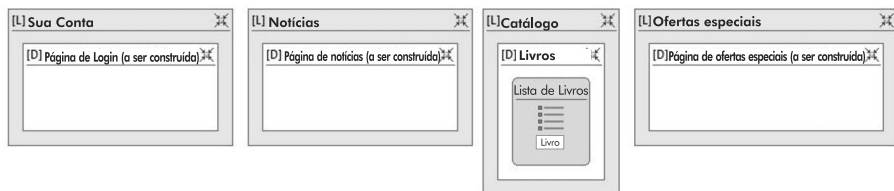


Figura 12.50

Exemplo de especificação de navegação em alto nível.

Sua Conta Notícias Catálogo Ofertas especiais			
> Catálogo > Livros			
Lista de Livros			
título	nomeDoAutor	preço	imagem DaCapa
➤ The Ultimate Hitchhicker's Guide	Douglas N. Adams	129	
➤ Dirk Getly's Holistic Detective Agency	Douglas N. Adams	6.99	
➤ Shave the Whales	Scott Adams	6.99	

Figura 12.51

Exemplo de renderização de uma página usando navegação de alto nível.

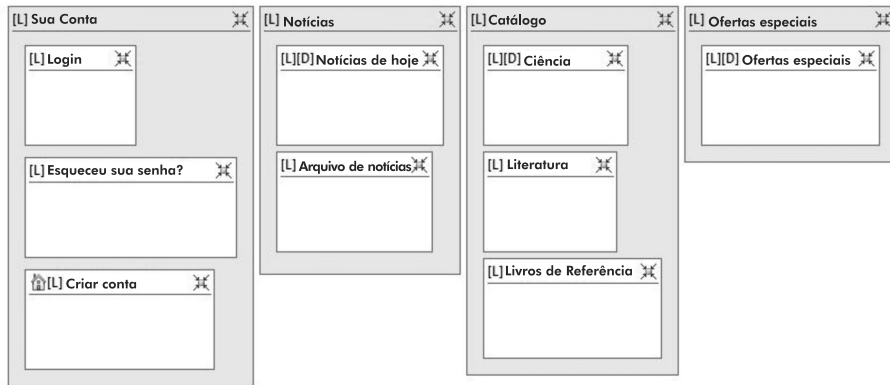
12.7.6 Navegação em alto nível

Navegação em alto nível é um padrão frequente em páginas web nas quais um menu de alto nível permite acesso a diferentes áreas do site. O exemplo na Figura 12.50 ilustra navegação em alto nível sendo definida por um conjunto de áreas *landmark*, cada uma contendo uma página (elas poderiam conter mais páginas, se necessário).

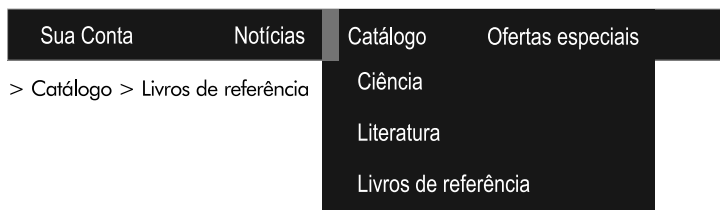
Para o exemplo, apenas a terceira área foi parcialmente definida. Uma renderização deste modelo é mostrada na Figura 12.51, em que a terceira área, “Catálogo”, é selecionada do menu de alto nível mostrando a página *default Livros*.

Uma variação desse padrão é a navegação de alto nível baseada em menus. Isso pode ser conseguido pela adição de um conjunto de páginas *landmark* a cada uma das áreas *landmark* de alto nível, como mostrado na Figura 12.52.

A Figura 12.53 mostra a renderização quando o mouse é posicionado sobre o item de menu “Catálogo”.

**Figura 12.52**

Exemplo de especificação de uma navegação baseada em menu de alto nível.

**Figura 12.53**

Exemplo de renderização de uma página usando navegação baseada em menu de alto nível.

**Figura 12.54**

Componentes de operação: *create*, *delete*, *update*, *connect*, *disconnect* e *reconnect*.

12.8 Modelagem de operações na interface

IFML permite a modelagem de operações básicas com o uso de *componentes de operação* que permite a criação, destruição, atualização, conexão, desconexão e reconexão⁷ de instâncias. Esses componentes realizam as operações básicas que foram apresentadas na Seção 8.5. A Figura 12.54 mostra a representação gráfica dessas operações.

⁷ Substituição de uma ligação.

Os componentes de operação não contêm nem renderizam informação: eles apenas a *processam*. Portanto, eles são representados graficamente fora das páginas.

Cada componente de operação deve ter pelo menos um fluxo de entrada. Algumas vezes, toda a informação que a operação precisa é obtida de um único componente de visão, mas, algumas vezes, mais do que um componente de visão pode ser necessário para fornecer todos os dados necessários para uma operação.

Componentes de operação têm fluxos de saída especiais. Dois tipos diferentes de fluxos de saída existem para operações:

- *Fluxo OK*, que define para onde vai o foco quando a operação foi executada com sucesso sobre todos os objetos.
- *Fluxo KO*, que define para onde vai o foco quando a operação falha para pelo menos um dos objetos afetados por ela.

Os componentes de operação são baseados em classes ou associações, as quais são suas fontes de dados. As operações *create*, *delete* e *update* são definidas sobre uma única classe, enquanto *connect*, *disconnect* e *reconnect* são definidas sobre uma associação e, portanto, têm uma classe *fonte* e uma classe-*alvo*.

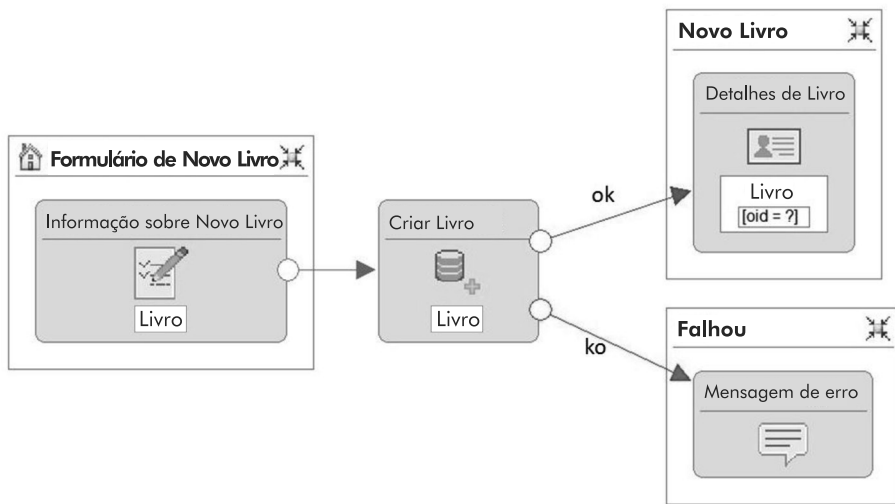
Por exemplo, uma operação *delete* poderia ser definida sobre a classe *Livro* como na Figura 12.54; esta operação seria responsável por deletar instâncias de *Livro*.

Uma operação *connect*, no entanto, poderia ser definida sobre uma associação, como *LivroParaEditora*, mostrado na Figura 12.54. Portanto ela seria responsável por conectar (adicionar ligações entre) instâncias da classe *Livro* (fonte) e a classe *Editora* (alvo).

12.8.1 Operação *create*

A operação *create* cria novas instâncias de uma dada classe. Um fluxo de navegação ou de dados pode ser usado para fornecer dados de inicialização para a nova instância. Usualmente, dados de inicialização são obtidos de um formulário baseado na mesma classe. Um fluxo do formulário para a operação *create* passa os valores necessários para todos os valores, exceto pelo OID, o qual é gerado automaticamente pela operação *create*. Por exemplo, a Figura 12.55 mostra uma estrutura para criar instâncias de *Livro*.

A Figura 12.56 mostra a renderização do formulário que está na origem do fluxo para a operação *create*. O fluxo é renderizado como o botão *Salvar*.

**Figura 12.55**

Exemplo de aplicação da operação *create*.

Informação do Novo Livro

isbn

título

nomeDoAutor

preço

númeroDePáginas

imagemDaCapa

Escolher arquivo

Nenhum Arquivo Selecionado

quantidadeEmEstoque

Salvar

Figura 12.56

Renderização de um formulário com um fluxo para uma operação *create*.

Como podemos ver, o usuário pode digitar no formulário valores de inicialização para um novo livro. O fluxo do formulário vincula dados e leva esses dados para a operação de criação. A operação de criação então cria a nova instância e define seus atributos com os valores recebidos do fluxo.

Se a operação tiver sucesso, então o fluxo OK é seguido para um componente de detalhes que mostra o livro que acaba de ser criado. O fluxo OK vincula automaticamente o OID do novo livro de forma que a expressão condicional no componente de detalhes pode ser usada para definir qual livro apresentar.

Se a operação falhar, o controle vai para uma página de erro, seguindo o fluxo KO.

12.8.2 Operação *delete*

A operação *delete* pode ser realizada sobre um ou mais objetos que satisfazem a expressão condicional que a define.

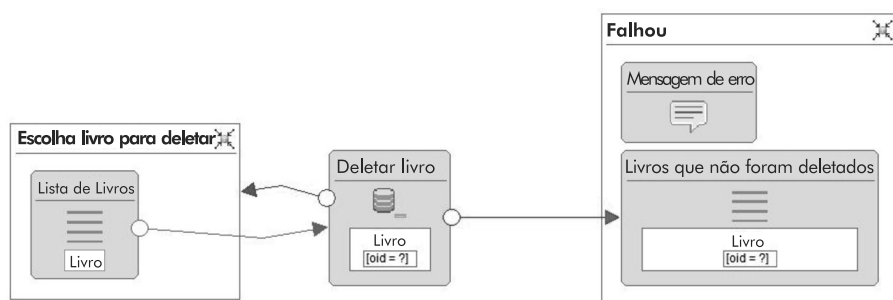


Figura 12.57

Uma operação *delete* sendo ativada a partir de uma lista.

> Escolha livro para deletar

Lista de Livros		
título	nomeDoAutor	
> Dirk Getty's Holistic Detective Agency	Douglas N. Adams	deletar
> Foundation and Empire	Isaac Asimov	deletar
> Rama II	Arthur C. Clarke and Gentry Lee	deletar
> Shave the Whales	Scott Adams	deletar
> The Hammer of God	Arthur C. Clarke	deletar
> The Long Dark Tea-Time of the Soul	Douglas N. Adams	deletar
> The Revenge of the Baby-Sat	Bill Waterson	deletar
> The Ultimate Hitchhicker's Guide	Douglas N. Adams	deletar

Figura 12.58

Renderização de uma lista simples associada a uma operação *delete* por um fluxo chamado “deletar”.

A operação *delete* pode ser usada em conjunto com uma lista (qualquer tipo de lista), na qual o usuário pode selecionar os objetos a serem deletados. A Figura 12.57 mostra um exemplo no qual o usuário seleciona um livro para deletar de uma lista simples. Note que o seletor da operação delete é baseado no OID do livro, o qual é recebido como um parâmetro do fluxo de entrada.

Esta página pode ser renderizada como na Figura 12.58, na qual a palavra “deletar” corresponde ao nome do fluxo.

Quando todos os objetos que se qualificam para ser deletados são realmente deletados, o fluxo OK é seguido. Quando pelo menos um objeto não puder ser deletado, o fluxo KO é seguido e passa como parâmetro vinculado os OIDs dos objetos que não foram deletados. Assim, o fluxo KO poderia, por exemplo, levar a uma lista ou um componente de múltiplos detalhes no qual os objetos que não puderam ser removidos são mostrados, como visto na Figura 12.57.

12.8.3 Operação *update*

Uma operação *update* contém um seletor para um ou mais objetos da mesma classe e um conjunto de atribuições para atualizar os atributos destes objetos. Usualmente, novos valores para atributos são recebidos por fluxos de navegação normais ou fluxos de dados.

A Figura 12.59 mostra como usar uma operação *update* para atualizar o preço de um livro existente.

Na página *Preços de Livros*, um livro é selecionado de uma lista (Figura 12.60). Então uma nova página (*Atualizar Preço*) mostra detalhes do livro e um campo pré-carregado com o preço antigo. O usuário pode mudar este valor e atualizá-lo pressionando “Salvar” (Figura 12.61).

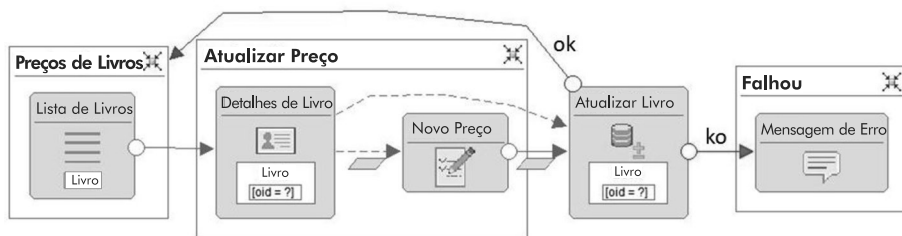


Figura 12.59

Uma operação *update* sendo usada para trocar o preço de um livro.

> Preços de Livros

Lista de Livros			
título		preço	
>	The Ultimate Hitchhiker's Guide	Douglas N. Adams	129 atualizar preço
>	Dirk Getly's Holistic Detective Agency	Douglas N. Adams	6.99 atualizar preço
>	Shave the Whales	Scott Adams	6.99 atualizar preço
>	The Long Dark Tea-Time of the Soul	Douglas N. Adams	6.99 atualizar preço
>	Rama II	Arthur C. Clarke and Gentry Lee	6.99 atualizar preço
>	The Hammer of God	Arthur C. Clarke	6.99 atualizar preço
>	Foundation and Empire	Isaac Asimov	5.99 atualizar preço
>	The Revenge of the Baby-Sat	Bill Waterson	8.95 atualizar preço

Figura 12.60

Renderização da página *Preços de Livros*.

> Atualizar Preço

Detalhes de Livro

títuloThe Revenge of the Baby-Sat
nomeDoAutorBill Waterson

Novo Preço

Novo Preço8.95

Salvar

Figura 12.61

Renderização da página *atualizar preço*.

O antigo preço é carregado no campo *Novo Preço* por um fluxo de dados de *Detalhes de Livro*. A operação *update* é ativada pelo fluxo de navegação que é renderizado como o botão *Salvar* na página *Atualizar Preço*. Este fluxo de navegação tem um parâmetro vin-

culado que passa o novo preço do livro do respectivo campo no formulário *Novo Preço* para a operação *update*. Outros atributos do livro, incluindo seu OID, são recebidos por um fluxo de dados de *Detalhes de Livro*.

Na Figura 12.59, o fluxo OK da operação *update* leva de volta para a lista na página *Preços de Livros*, na qual o novo preço pode ser visualizado. O fluxo KO leva para uma janela de erro. Novamente, o fluxo OK é seguido quando todos os objetos qualificados pela expressão condicional são atualizados. Se pelo menos um objeto não puder ser atualizado, o fluxo KO é seguido com o conjunto de OIDs dos objetos que não puderam ser atualizados como parâmetros vinculados.

12.8.4 Operações *connect*, *disconnect* e *reconnect*

As operações *connect*, *disconnect* e *reconnect* têm uma associação como fonte de dados. As operações *connect* e *disconnect* têm duas expressões condicionais: uma para o papel de origem e outra para o papel-alvo da associação. A operação *reconnect* pode ser entendida como uma combinação das anteriores. Ela tem três expressões condicionais: uma para o papel de origem, outra para os objetos no papel-alvo que vão ser desconectados, e uma terceira para os objetos no papel-alvo que vão ser conectados aos objetos da origem.

As operações *connect*, *disconnect* e *reconnect* podem adicionar, remover e substituir ligações entre conjuntos de objetos. Se, por exemplo, cinco objetos satisfazem a expressão condicional da origem de um *connect* e três objetos satisfazem a expressão condicional de destino, então uma operação de conexão iria criar 15 ligações: cada um dos cinco objetos na origem seria ligado a cada um dos três objetos no alvo.

O exemplo da Figura 12.62 mostra como uma ligação entre um livro e uma editora pode ser criada. Primeiramente, a ligação é obrigatória do livro para a editora. Assim, não é possível simplesmente pegar uma lista de livros e uma lista de editoras e escolher quais serão ligados. Uma ligação entre um livro e uma editora deve ser criada assim que o livro for criado. Além disso, essa ligação é imutável e não poderá mudar mais tarde.

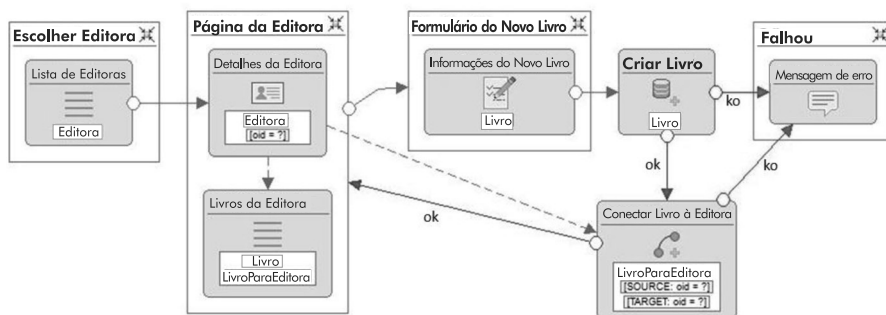


Figura 12.62

Modelo IFML mostrando uma interface na qual livros podem ser criados e ligados a editoras.

[Novo Livro](#)

> Página da Editora

Detalhes da Editora

nome

Bantam

município

New York

Livros da Editora

titulo	nomeDoAutor
> Dirk Getty's Holistic Detective Agency	Douglas N. Adams
> Foundation and Empire	Isaac Asimov
> The Hammer of God	Arthur C. Clarke

Figura 12.63

Renderização da *Página da Editora*.

Na primeira página *Escolher Editora*, existe uma lista simples de editoras. O usuário deve escolher uma e ser levado para a *Página da Editora*. Lá, os detalhes da editora e uma lista de livros já ligados a ela são mostrados. Note que *Detalhes da Editora* tem uma condição-chave e um fluxo de *Lista de Editoras* e, portanto, ela mostra a editora selecionada na lista. A lista *Livros da Editora* tem um fluxo de dados que vem de *Detalhes da Editora* e uma expressão condicional baseada em papel, de forma que apenas livros que são ligados à editora de *Detalhes da Editora* são mostrados. A Figura 12.63 mostra a renderização de *Página da Editora* após a editora “Bantam” ser selecionada da lista.

A partir da *Página da Editora*, o usuário pode navegar para o *Formulário de Novo Livro* usando um fluxo de navegação normal entre ambas as páginas, o qual é renderizado como a ligação *Novo Livro*. Nessa página, o usuário pode preencher o formulário com informações sobre o novo livro, o qual será automaticamente ligado à editora apresentada na *Página da Editora*. Inicialmente, a operação *create* cria uma nova instância de livro. Seu fluxo OK leva para uma operação *connect* que é baseada na associação *LivroParaEditora*. O fluxo que sai da operação *create* tem a chave do novo livro como parâmetro vinculado; esta chave é usada na expressão condicional do papel fonte. A editora é o papel alvo e ela é obtida pelo fluxo de dados que vem de *Detalhes da Editora*.

Se a operação de conexão for um sucesso, ela retorna o foco para a *Página da Editora*, na qual o novo livro pode ser visto na lista de livros da editora corrente. Se a operação *create* ou *connect* falhar, então o foco vai para uma página com uma mensagem de erro.

Entretanto, o design da Figura 12.62 não é seguro. Se um livro for criado e a operação de conexão falhar, este livro ficará inconsistente em relação à sua definição. IFML permite que uma transação seja definida como um grupo de operações. A Figura 12.64 mostra um

grupo de operações sendo definido para incluir tanto a operação *create* quanto a operação *connect*. Como o grupo de operações pode ser definido como uma transação ([T]), então ou todas as operações dentro do grupo obtêm sucesso, ou o grupo todo falha. No caso, um único fluxo KO saindo do grupo seria seguido.

12.9 Modelos IFML para operações CRUD

Esta seção mostra passo a passo como implementar uma interface no estilo CRUD para gerenciar informação sobre editoras que usam componentes de visão e operações IFML.

A página principal do CRUD *Gerenciar Editoras* apresenta uma lista de editoras existentes. A criação de uma nova editora é iniciada por um clique em uma ligação na página principal que dá acesso a uma nova página com um formulário. Se a criação de uma nova editora tiver sucesso, o controle retorna para a página principal, e a nova editora aparece na lista. Se houver um erro, o controle vai para uma página de erro, a partir da qual é possível retornar à página principal (Figura 12.65).

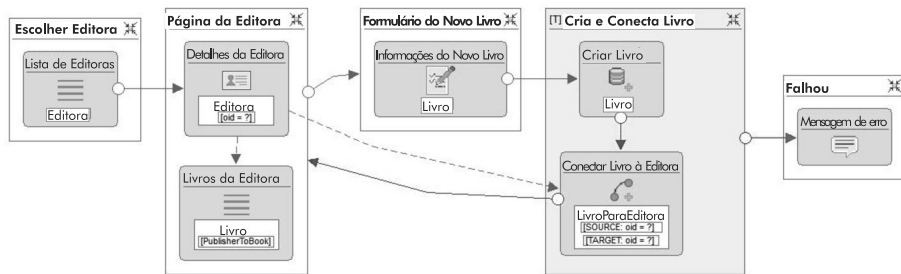


Figura 12.64

Um design mais seguro para o modelo da Figura 12.62 usando um grupo de operações.

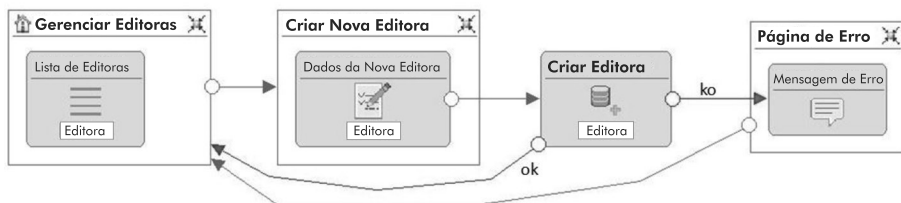
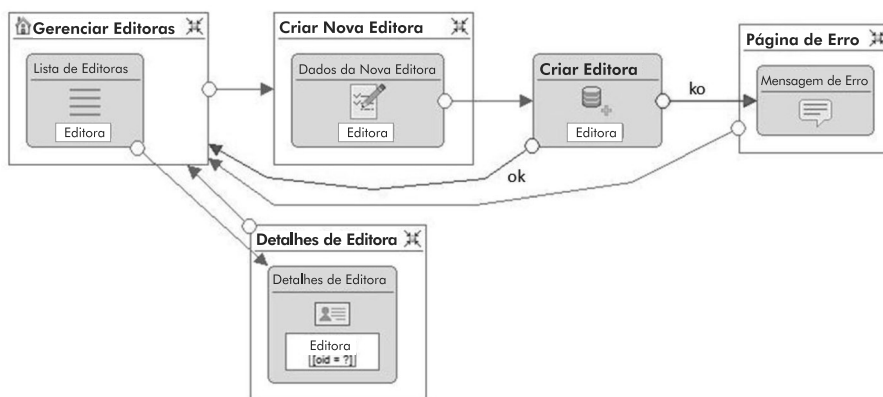
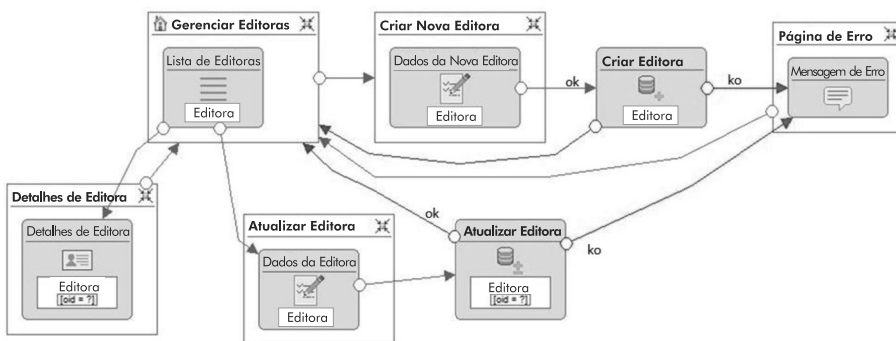


Figura 12.65

Definição de uma interface para *create*.


Figura 12.66

Create e retrieve definidos.

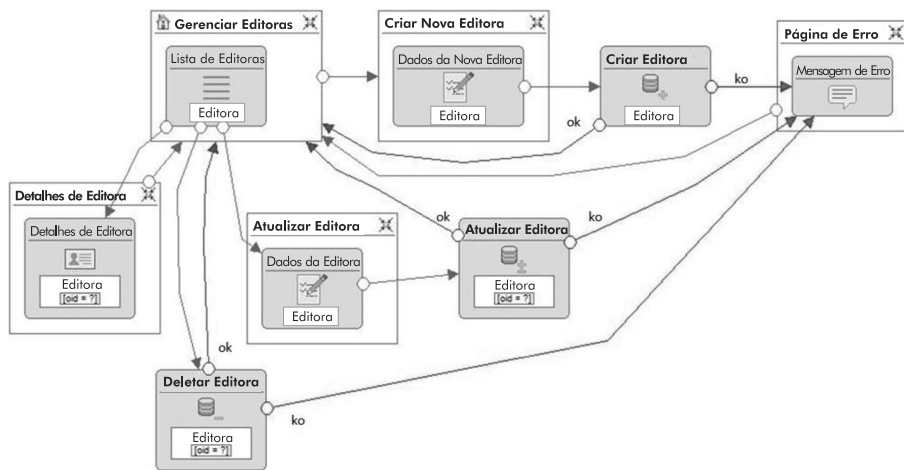

Figura 12.67

Create, retrieve e update definidos.

A segunda operação de um CRUD, *retrieve*, é uma consulta que permite a visualização de todos os atributos de uma dada editora. Esta consulta pode ser modelada por um fluxo da lista principal levando para uma página com um componente de visão de detalhes, como mostrado na Figura 12.66.

A terceira operação, *update*, permite que uma editora seja selecionada na lista principal, e os atributos da editora selecionada são usados para preencher os campos de um formulário com *Editora* como classe fonte. Ali eles podem ser editados pelo usuário e salvos, conforme mostrado na Figura 12.67.

Finalmente, a operação para deletar uma editora é adicionada, como mostra a Figura 12.68. Uma operação de deleção pode ser acessada a partir da lista principal.

**Figura 12.68**

Especificação completa de uma interface CRUD.

**Figura 12.69**

Janela principal para o CRUD de *Editora*.

A Figura 12.69 mostra a renderização da página principal do CRUD de *Editora* como definido na Figura 12.68. No topo da página, é possível ver a ligação para a criação de uma nova editora, e logo a seguir está a lista principal, na qual as opções para consulta, atualização e deleção de cada editora estão disponíveis.

A Figura 12.70 mostra a renderização para a página de consulta, a qual é acessada pela seleção da ligação *detalhes* para a terceira editora na lista da página principal. A ligação *retornar* no topo da página permite a um usuário voltar para a página principal.

A Figura 12.71 apresenta a página de atualização da editora, a qual é acessada pelo clique na ligação *atualizar* na lista da página principal. Esta página é idêntica à página para criação de uma nova editora, exceto pelo fato de que ela pré-carrega dados em vez de estar em branco e que, como o atributo *nome* é imutável, seu respectivo campo é definido como não modificável.

12.10 Modelagem de interfaces de casos de uso com IFML

Conforme visto na seção anterior, trabalhar com as operações básicas neste nível pode rapidamente tornar o diagrama muito complexo para ser facilmente entendido. Uma opção para lidar com essa complexidade é separar o projeto da interface em diferentes áreas e lidar com uma de cada vez. Isso já é possível, porque os casos de uso já foram explorados e a equipe já tem um entendimento sobre a estrutura do sistema e das necessidades do usuário, o que permite que a equipe faça um design aceitável para a interface.



Figura 12.70

Renderização para a página de consulta (*retrieve*) de editora.



Figura 12.71

Renderização para a página de atualização de editora.

O design da interface com usuário pode iniciar com o diagrama de caso de uso da Figura 3.11, parte do qual é apresentado na Figura 12.72.

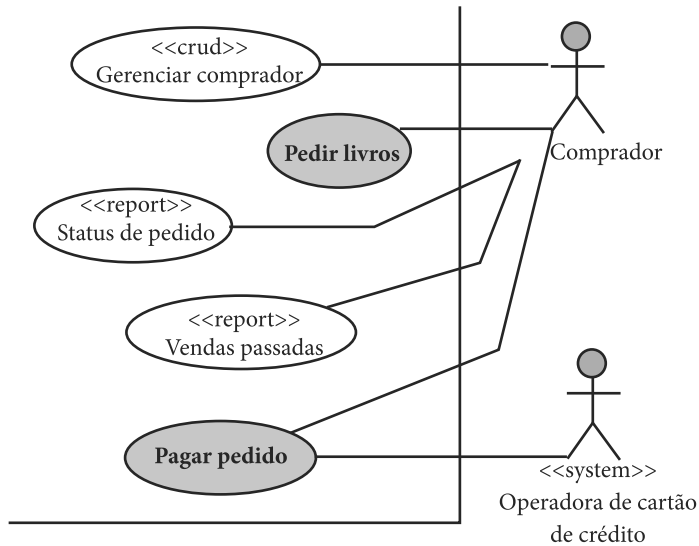


Figura 12.72

Diagrama de casos de uso parcial de Livir com os casos de uso do comprador.

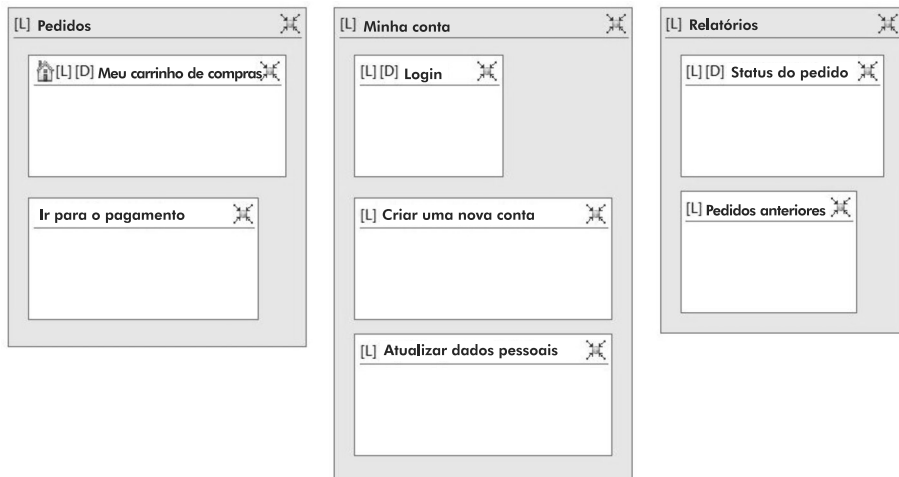


Figura 12.73

Design inicial de alto nível para a visão de site de Livir.

Baseando-se neste diagrama de caso de uso, um possível design de interface que poderia ser preparado teria três áreas principais:

- Carrinho de compras: pedidos e pagamento.
- Minha conta: login e dados pessoais.
- Relatórios.

Este é ainda um design bastante cru, no sentido de que nenhuma questão de usabilidade foi considerada para decidir sobre a melhor organização possível. Entretanto, ele é efetivo. A Figura 12.73 mostra o design IFML dessas áreas.

Agora vamos olhar para o caso de uso 01: *Pedir livros*. Seu diagrama de sequência de sistema é mostrado na Figura 12.74:

Para preparar um design de interface baseado nesse diagrama de sequência de sistema, a equipe deve olhar para as operações de sistema, inclusive seus parâmetros e resultados:

- Qualquer operação de sistema, seja ela um comando ou consulta, deve ser ativada em algum lugar da interface. A ativação pode ser obtida pelo clique em um botão ou ligação ou pela seleção de um item em uma lista etc. Algumas vezes, o mesmo evento pode ativar mais do que uma operação de sistema: se duas ou mais operações de sistema acontecem imediatamente após um evento de sistema (lado esquerdo do diagrama de sequência), então elas possivelmente são ativadas pela mesma ação do usuário.

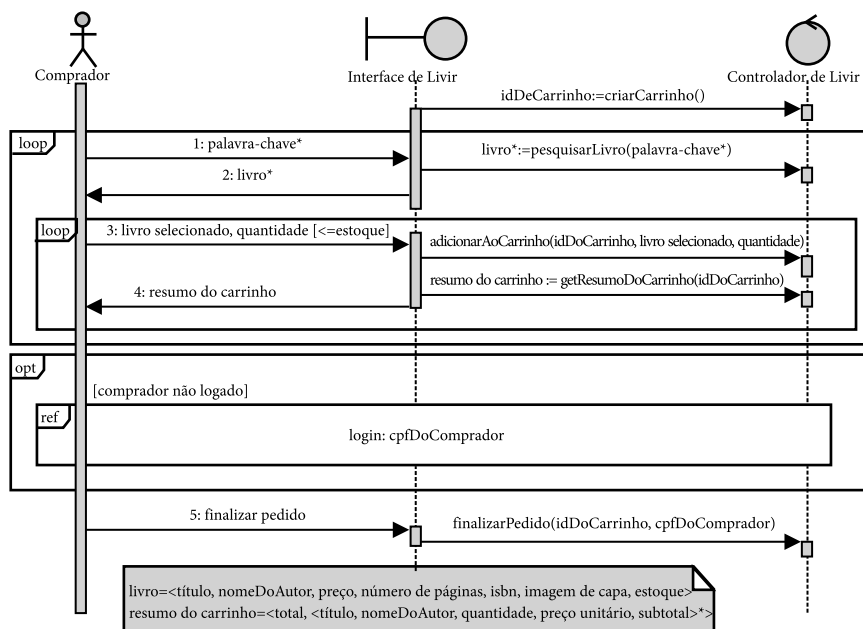
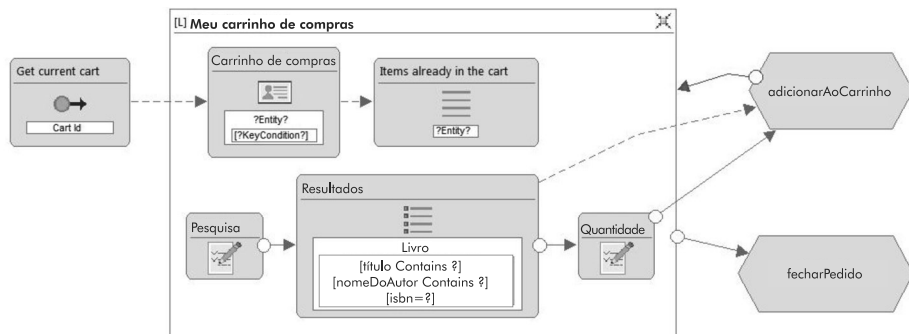


Figura 12.74

Diagrama de sequência de sistema para o caso de uso 01: *Pedir livros*.

**Figura 12.75**

Refinamento do design de interface para a página *Meu carrinho de compras*.

- Qualquer parâmetro de uma operação de sistema deve ser obtido em algum lugar. Alguns parâmetros poderiam ser obtidos de outras fontes (um relógio, por exemplo), mas a maioria dos parâmetros seria obtido na interface. Assim, para a maioria dos parâmetros, haverá um componente de visão na interface (usualmente um campo de formulário) que permite que o usuário introduza os dados necessários.
- Qualquer resultado obtido por uma consulta de sistema deve ser apresentado em algum lugar. Alguns resultados podem estar escondidos do usuário, em alguns casos, mas usualmente eles são apresentados. Componentes de visão, como listas, detalhes e outros, podem ser usados para mostrar os resultados de consultas de sistema.

Para proceder com o design de interface, as recomendações anteriores são seguidas e os componentes de visão são criados para conter todos os parâmetros e resultados necessários. Ações de operações de sistema (representadas como hexágonos) são adicionadas ao diagrama também. A Figura 12.75 apresenta a evolução do design de interface para a página *Meu carrinho de compras*, a qual implementa parte do caso de uso 01: *Pedir livros*.

A operação de sistema *criarCarrinho* produz um novo carrinho que é apresentado pelo componente de detalhes *Carrinho de Compras*. O usuário pode usar o formulário *Pesquisar* para pesquisar por livros, os quais são representados na lista *Resultados de Pesquisa*. Lá, o usuário pode selecionar um livro e definir a quantidade desejada no formulário *Quantidade*. Isso ativa a operação de sistema *adicionarAoCarrinho*. Essa operação toma parâmetros de *Carrinho de compras* (o id do carrinho), *Resultados de Pesquisa* (o *isbn*) e *Quantidade* (a *quantidade*). Após um livro ser inserido no carrinho, ele é automaticamente mostrado no componente de detalhes múltiplos *Resumo do Carrinho*.

Antes de finalizar o pedido, o comprador deve se logar, caso ainda não o tenha feito. Finalmente, o pedido pode ser finalizado pela execução da operação de sistema *finalizarPedido*, a qual toma parâmetros de *Comprador* (o *cpf* do comprador) e *Carrinho de Compras* (o id do carrinho).

Assim, em vez de usar operações IFML para definir operações de sistema, a equipe pode usar procedimentos definidos externamente, como na Figura 12.75. Nesse caso, a equipe pode implementá-los em sua linguagem de programação favorita.

12.11 O processo visto aqui

	Concepção	Elaboração
Modelagem de negócio		
Requisitos		
Análise e design		Fazer o design da camada de interface: • Fazer o design da organização da interface em nível mais alto baseado em áreas e páginas. • Refinar páginas, incluindo novos componentes para parâmetros ou operações de sistema e resultados para consultas de sistema. • Associar a ativação de operações com eventos de interface, como o acionamento de botões ou menus.
Implementação		
Teste		
Gerenciamento de Projeto		

12.12 Questões

1. Escolha pelo menos três casos de uso do sistema *Livir* (Tabela 4.7): um CRUD, um relatório e um caso de uso não estereotipado. Crie um modelo de interface completo para eles.
2. Pesquise fontes bibliográficas sobre os padrões de design *Model-View-Controller* (MVC) e *Model-View-Presenter* (MVP). Quantas diferentes definições e variações para esses padrões você é capaz de achar? Como elas diferem uma da outra?

Persistência de dados

13

TÓPICOS-CHAVE NESTE CAPÍTULO

- Mapeamento Objeto-Relacional (ORM)
- Proxy virtual
- Brokers
- Caches virtuais

13.1 Introdução à persistência de dados

A disponibilidade de mecanismos de persistência para linguagens comerciais¹ tem tornado o design do banco de dados muito simples para vários projetos. Com ferramentas adequadas, é possível gerar automaticamente a camada de persistência para um grande número de sistemas de informação. Para alguns sistemas críticos e legados, entretanto, ajustes em bancos de dados podem ainda ser necessários para acomodar características especiais ou para satisfazer requisitos de performance ou segurança.

Usualmente, sistemas orientados a objetos são implementados em linguagens orientadas a objetos, mas o armazenamento de dados persistentes² é realizado com *bancos de dados relacionais*. Embora outras técnicas, tais como bancos de dados orientados a objetos (Won, 1990) e bancos de dados XML (Bourret, 2010) também sejam opções para implementar armazenamento permanente de dados, o padrão *ORM*, ou *mapeamento objeto-relacional* (*object-relational mapping*), é ainda a abordagem preferida.

O objetivo deste capítulo é explicar o que acontece dentro da camada de persistência de um sistema quando um mecanismo de persistência baseado em *ORM* é usado. Primeiramente, um bom mecanismo de persistência requer que o domínio e o armazenamento de dados sejam separados em diferentes camadas dentro da aplicação. Lembre que a camada de interface é modelada com casos de uso essenciais e a camada de domínio é projetada usando o modelo conceitual e os contratos como base. Nenhuma das camadas antes mencionadas trata de armazenamento de dados ou persistência. Persistência poderia ser projetada como uma questão separada: um mecanismo de segundo plano que vai manter os dados segura e permanentemente em seu lugar sem interferir na lógica de domínio ou interface.

¹ Veja, por exemplo: <<http://www.hibernate.org/>>.

² *Persistente*, neste contexto, é o oposto de *temporário*, ou seja, informação persistente é a informação que deve ser mantida até que algum usuário explicitamente a delete. Informação temporária é mantida apenas durante uma sessão de uso do sistema.

O mecanismo de persistência deveria assegurar que os objetos sejam salvos em um dispositivo de memória permanente, e que eles sejam carregados de lá quando necessário. As camadas de domínio e interface com usuário não deveriam ser poluídas com questões de persistência.

O design orientado a objetos fornece muitas boas práticas, tais como encapsulamento, responsabilidades e delegação, o que ajuda os designers a lidar com as complexidades da lógica para acessar e transformar a informação. Mas quando os dados devem ser armazenados em um meio mais permanente – discos e fitas, por exemplo –, o banco de dados relacional é uma boa opção porque ele é muito eficiente em termos de desempenho em relação ao tempo, e existem muitas técnicas de otimização para melhorar as operações relacionais. Assim, se a equipe quiser trabalhar no melhor dos dois mundos (objeto e relacional), o mapeamento entre eles precisa ser compreendido.

A literatura se refere ao problema do *conflito de impedância* objeto-relacional³ (Ireland; Keynes; Bowers; Newton; Waugh, 2009) porque a maioria das boas características obtidas pelo design orientado a objetos é perdida quando um banco de dados relacional plano é usado.

A implementação de um mecanismo de persistência minimiza este problema pelo uso de um conjunto de classes predefinidas que não pertencem à camada de domínio e que cuidam de toda a lógica que envolve objetos de domínio sendo salvos e recuperados de um banco de dados relacional.

13.2 Mapeamento Objeto-Relacional (ORM)

Um diagrama de classes de design (DCD) completo e detalhado permite a geração automática de uma estrutura de banco de dados relacional que reflete em memória secundária⁴ a informação que os objetos representam em memória principal. As seções seguintes apresentam algumas regras de equivalência que deveriam ser observadas quando se usa ORM.

13.2.1 Classes e atributos

O primeiro conjunto de regras trata classes e seus atributos. Cada classe persistente do DCD corresponde a uma *tabela relacional*. Cada atributo é uma *coluna* da tabela e cada instância é uma *linha* ou *registro* da tabela.

Os estereótipos de alguns atributos, como <<unique>>, <<optional>> e <<immutable>> afetam as propriedades das colunas das tabelas relacionais. Algumas das caracterís-

³ Object-relational impedance mismatch.

⁴ A *memória secundária* é usualmente bem mais lenta do que a *memória principal*. Entretanto, ela é também muito mais barata e, assim, é largamente usada para armazenar quantidades de dados que não caberiam nas memórias principais muito mais caras. Usualmente, a *memória secundária* consiste em mídia como fitas ou discos óticos ou magnéticos. Dados armazenados em *memória secundária* usualmente não podem ser processados de forma direta pelo computador e precisam ser carregados na *memória principal*. A *memória principal* é também conhecida como *Random Access Memory* (RAM) e fisicamente ela é implementada de maneira usual em circuitos integrados eletrônicos.

ticas descritas aqui podem não ser implementadas por alguns sistemas gerenciadores de bancos de dados comerciais. Nesse caso, ajustes podem ser necessários para fornecer uma implementação segura e livre do conflito de impedância objeto-relacional.

Atributos estereotipados com `<<unique>>` são representados como colunas marcadas com *unique* ou *uniq*, as quais não podem repetir valores. Entretanto, mesmo objetos sem atributo único tem uma *identidade* que os distingue de outros objetos. Até objetos com o mesmo valor para cada um dos atributos podem ser diferenciados pela sua identidade. No caso das tabelas relacionais, isso é implementado pela definição de uma *chave primária* para cada tabela relacional. Uma chave primária, assim como uma coluna *unique*, não pode repetir elementos. Um elemento em uma coluna de chave primária também nunca pode ser nulo. Chaves primárias usualmente são simples sequências numéricas geradas automaticamente pela aplicação de forma que o mesmo número nunca seja gerado duas vezes para a mesma tabela. *Chaves primárias não correspondem a qualquer dos atributos de um objeto*: elas correspondem à identidade do objeto.

A Figura 13.1 mostra a classe que é a base para os exemplos seguintes, e a Tabela 13.1 mostra a tabela relacional equivalente com três instâncias da classe representada.

Livro
<code><<immutable>> <<unique>> +isbn:ISBN</code> <code><<immutable>> +título:String</code> <code><<immutable>> +nomeDoAutor:String</code> <code>+preço:Moeda</code> <code><<immutable>> +númeroDePáginas:Natural</code> <code>+ /nomeDaEditora:String=editora.nome</code> <code><<optional>> +imagemDaCapa:Imagem</code> <code>+quantidadeEmEstoque:InteiroNãoNegativo=0</code>

Figura 13.1

Classe de referência.

Tabela 13.1 Tabela relacional equivalente à classe da Figura 13.1.							
Tabela: Livro							
PK, Uniq, Im, NN	Uniq, Im, NN	Im, NN	Im, NN	NN	Im, NN		NN
pkLivro	isbn	título	nomeDoAutor	preço	número DePáginas	imagem DaCapa	quantidade EmEstoque
10001	0553286587	Rama II	Arthur C. Clarke and Gentry Lee	6,99	466		2
10002	0553293370	Foundation and Empire	Isaac Asimov	5,99	282		3
10003	0671742515	The Long Dark Tea Time of the Soul	Douglas N. Adams	6,99	307		21

Embora outros formatos possam ser usados para especificar restrições nas colunas, aqui eles são apresentados como acrônimos para rápida referência:

- *Uniq*, ou *unique*, significa que a coluna não pode repetir valores.
- *Im*, ou *immutable*, significa que o valor na coluna não pode ser atualizado.
- *NN*, ou *not null*, significa que a coluna não admite o valor nulo. Isto é exatamente o oposto do estereótipo <<*optional*>> usado para o design de classes. Ele é usado aqui porque *not null* é uma restrição comumente implementada em gerenciadores de bancos de dados⁵.
- *PK*, ou *primary key*, significa que a coluna é a chave primária da tabela. Se uma coluna é *PK*, ela é necessariamente imutável e não nula. Se apenas uma única coluna é *PK*, então ela deve ser *unique* também. Entretanto, se a *PK* é composta, se estendendo sobre mais do que uma coluna, então as colunas individuais não necessariamente devem ser *unique*.

A primeira coluna da Tabela 13.1 é *pkLivro*. Note que ela não corresponde a nenhum dos atributos da classe de referência. Seu valor é artificialmente gerado por um *gerador de sequência* de forma que ele seja único para toda a aplicação ou pelo menos para aquela tabela. Alguns autores (Wieringa; Jonge, 1991) referem-se a isso como *chave substituta*⁶, ou seja, um valor que não tem significado semântico, e que é único no sistema, nunca reusado, gerado pelo sistema e não manipulado pelo usuário ou aplicação.

As outras colunas correspondem aos atributos da classe de referência:

- *isbn* é um atributo que não admite repetição. Portanto, a coluna correspondente é *unique*, imutável e não nula.
- *título*, *nomeDoAutor* e *númeroDePáginas* são atributos normais que são imutáveis e obrigatórios. Portanto, as colunas equivalentes são imutáveis e não nulas, mas não *unique*.
- *preço* e *quantidadeEmEstoque* são atributos normais que podem ser atualizados, mas não podem ser nulos. Portanto, a coluna equivalente é apenas não nula.
- *imagemDaCapa* é um atributo opcional que pode ser atualizado. Portanto, a coluna respectiva não tem restrição: ela pode ser atualizada e pode ser nula.
- *nomeDaEditora* é um atributo derivado. Portanto, ele não é representado na tabela relacional.

Se um atributo de classe de design for marcado com <<*transient*>>, então ele também não deve aparecer na tabela relacional.

⁵ Via de regra, não se recomenda que células de bancos de dados relacionais contenham o valor null porque isso aumentaria a complexidade do código que faria acesso a estas células (Date, 1982). Assim, campos textuais usualmente usam a string vazia no lugar do valor null, e outros tipos de campos usam valores especiais que indicam a ausência de valor. Não estamos recomendando aqui que tabelas devam utilizar o valor null; a ausência da condição *nn* em uma coluna indica que o campo admite que falte um valor ali. O responsável pelo projeto do banco de dados é que irá decidir e arcar com as consequências de usar o valor null diretamente ou valores substitutos.

⁶ *Surrogate key*.

13.2.1.1 Gerador de sequência numérica

A maioria dos sistemas gerenciadores de bancos de dados fornece geradores de sequência numérica que geram números que nunca se repetem. Isso é necessário para fornecer valores únicos para chaves primárias, especialmente quando múltiplos usuários estão produzindo novos registros ao mesmo tempo. O gerador de sequência numérica deve assegurar que diferentes usuários não possam produzir o mesmo número.

Os geradores de sequência numérica podem ser associados diretamente à coluna da tabela que contém chaves primárias. Cada vez que um novo registro é inserido na tabela, um novo número na sequência é criado.

O designer do banco de dados pode escolher um valor inicial para a sequência, bem como o seu incremento. Por exemplo, iniciando em 500 com um incremento de 5, a sequência gerada seria 500, 505, 510, 515 etc.

13.2.1.2 Seleção de índice

Por *default*, uma tabela relacional é apenas um conjunto de registros. Encontrar um objeto em uma tabela poderia requerer fazer uma iteração sobre todos os elementos até que o elemento desejado fosse encontrado. Por exemplo, procurar por um determinado livro, dado o seu ISBN, iria requerer uma busca exaustiva.

Bancos de dados usualmente fornecem, entretanto, a possibilidade de indexar colunas. Uma *coluna indexada* tem uma tabela auxiliar que permite encontrar registros específicos em tempo quase constante. Por exemplo, se a coluna *isbn* da tabela for indexada, então quando um livro for procurado baseado em seu ISBN, nenhuma iteração será realizada sobre o conjunto de todos os registros: o sistema simplesmente vai traduzir o valor do ISBN em uma posição de memória, usando uma função de espalhamento⁷, e recuperar o elemento daquela posição. Se este não for o elemento desejado, então ele vai olhar o seguinte, e assim por diante, até encontrar o elemento desejado. Se a função de espalhamento for bem implementada e a tabela de espalhamento tiver espaço suficiente para evitar colisões (dois valores sendo traduzidos para o mesmo valor de espalhamento), então usualmente o elemento desejado está realmente no primeiro lugar pesquisado, ou muito próximo dele pelo menos.

O uso de índices melhora a velocidade das consultas. Entretanto, ele retarda a atualização do banco de dados porque toda vez que um registro for atualizado, inserido ou deletado, a tabela auxiliar também precisa ser atualizada (Choenni; Blanken; Chang, 1993). Além disso, índices também requerem mais espaço de armazenamento para acomodar as tabelas auxiliares.

Uma chave primária é indexada por *default*. As outras colunas podem ser indexadas se o designer escolher fazer isso. Dadas as restrições mencionadas acima, criar outros índices poderia ser uma vantagem no caso de um atributo que é usado como qualificador

⁷ Hash.

interno. Nesse caso, encontrar objetos rapidamente pode ser crucial para o desempenho da aplicação. Caso contrário, índices deveriam ser evitados. Por exemplo, colunas que raramente são usadas com propósito de pesquisa (por exemplo, o número de páginas do livro) não deveriam ser indexadas.

13.2.2 Associações

Geralmente, associações entre classes (exceto associações temporárias que não persistem) correspondem a *tabelas associativas* no modelo relacional, ou seja, tabelas com uma chave primária composta dos valores das chaves primárias das tabelas que representam as classes participantes.

Nesse caso, a chave primária da tabela associativa é de fato composta de duas (ou mais) colunas. Cada coluna poderia repetir valores individualmente dependendo da multiplicidade de associação; mas o par (ou tupla) de valores que compõe a chave primária nunca poderia ser repetido.

Dependendo da multiplicidade dos papéis de associação, algumas regras devem ser observadas. Associações “muitos para muitos” não têm restrições *unique* em colunas individuais. Entretanto, associações “um para muitos” e “um para um” requerem que uma ou ambas as colunas da chave primária composta sejam *unique*. Isso é explicado em maior detalhe nas subseções seguintes.

13.2.2.1 Associações “muitos para muitos”

Se a associação é de “muitos para muitos” com as multiplicidades dos dois papéis definidas como *, então não há restrições nas colunas que compõem a chave primária. A Figura 13.2 mostra um exemplo de associação de muitos-para-muitos.

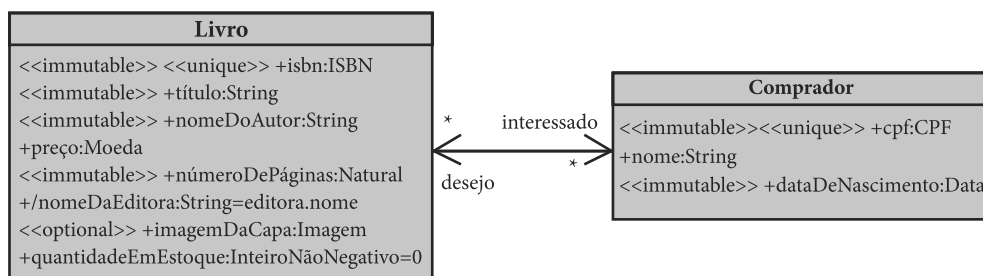


Figura 13.2

Associação “muitos para muitos”.

Considere novamente os três livros da Tabela 13.1 e os três compradores mostrados na Tabela 13.2.

Note que na Tabela 13.2 um comprador tem dois códigos: um é seu *cpf*⁸: este valor é conhecido fora do sistema de informação. Ele é usualmente preformatado e, embora em condições normais ele não mude, ninguém pode evitar que o Governo eventualmente mude os números de CPF de todos os cidadãos, se necessário. Portanto, embora o *cpf* seja único para o comprador, ele não se qualifica como uma boa chave primária. A chave primária para a Tabela 13.2 é *pkComprador*, a qual é um número criado automaticamente por um gerador de sequência de números; ele é único e garante-se que nunca mudará porque ele não tem significado fora do banco de dados.

A Tabela 13.3 mostra uma tabela relacional associativa que representa as ligações entre alguns compradores e alguns livros desejados por eles. Note que a *PK* se estende sobre duas colunas agora. Cada coluna contém uma *chave estrangeira* (*FK*⁹), ou seja, um valor que corresponde à chave primária de outra tabela.

Na tabela associativa *interessado_desejo*, as colunas *fkComprador* e *fkLivro* juntas formam a *chave primária composta*. Ambas não podem ser nulas, e pares *fkComprador/fkLivro* não podem ser repetidos. Entretanto, como a associação é de muitos para muitos, cada coluna individual pode repetir valores, como visto na tabela. As colunas individuais seriam imutáveis apenas se um ou ambos papéis da associação o fossem.

Tabela 13.2 Tabela relacional para a classe *Comprador*

Tabela: Comprador			
PK, Uniq, Im, NN	Uniq, Im NN	NN	Im, NN
<i>pkComprador</i>	<i>cpf</i>	<i>nome</i>	<i>dataDeNascimento</i>
20001	111.111.111-11	Abe	04/01/1970
20002	222.222.222-22	Beth	23/02/1982
20003	333.333.333-33	Charles	05/12/1979

Tabela 13.3 Tabela associativa que representa uma associação “muitos para muitos”.

Tabela: interessado_desejo	
PK	
NN	NN
<i>fkComprador</i>	<i>fkLivro</i>
20001	10001
20002	10001
20001	10003

⁸ Os valores de CPF usados na tabela são propositalmente impossíveis e inválidos para evitar o uso acidental de algum valor que corresponda a uma pessoa verdadeira.

⁹ *Foreign Key*.

A Tabela 13.3 mostra que Abe (comprador 20001) deseja os livros “Rama II” (10001) e “The Long Dark Tea-Time of the Soul” (10003). Beth (comprador 20002) apenas deseja “Rama II” (10001), e Charles (20003) não tem desejos.

Em vez de nomear a tabela associativa com os nomes das classes (*Comprador_Livro*), é preferível nomeá-la com os nomes dos papéis (*interessado_desejo*), porque mais do que uma associação pode existir entre duas classes.

Se a associação for obrigatória em uma ou ambas as direções, então considerações especiais devem ser observadas:

- Se a associação é obrigatória em *uma* direção, então todas as instâncias do *outro* lado devem aparecer pelo menos uma vez na tabela associação. Se no exemplo da Figura 13.2, o papel *interessado* fosse obrigatório (1..*), então cada livro deveria ter pelo menos um comprador associado. Então, cada valor de chave primária da tabela *Livro* deveria aparecer na tabela *interessado_desejo* pelo menos uma vez.
- Se a associação for obrigatória em *ambos* os lados, então todas as instâncias de ambas as classes devem aparecer pelo menos uma vez na tabela associativa. Se no exemplo da Figura 13.2, ambos os lados fossem obrigatórios (1..*), então cada livro da tabela *Livro* e cada comprador da tabela *Comprador* deveriam aparecer pelo menos uma vez na tabela associativa.

De forma mais geral, considerando que *A* tem uma associação com *B*, e que o limite inferior do papel *B* é *n* enquanto o limite superior é *m* (multiplicidade *n..m*), o número de vezes que cada instância de *A* deve aparecer na tabela associativa é no mínimo *n* e no máximo *m*. Por exemplo, se a multiplicidade do papel *B* fosse 2..5, então cada instância de *A* deveria aparecer na tabela associativa pelo menos duas vezes e não mais do que cinco vezes. Infelizmente, este tipo de restrição usualmente não está presente nos sistemas gerenciadores de bancos de dados.

13.2.2.2 Associações “um para muitos” e “muitos para um”

Quando a associação é de “um para muitos”, então a coluna do lado *muitos* deve ter uma restrição *unique*. Isso significa que a coluna não pode repetir elementos individualmente, enquanto a outra coluna pode repeti-los. Os elementos que não podem ser repetidos na tabela associativa, portanto, podem ser ligados a um único elemento na outra tabela; esta restrição garante que a associação seja de “um para muitos”. A Figura 13.3 mostra um exemplo de uma associação “um para muitos”.

A Tabela 13.4 mostra uma tabela associativa para a associação “um para muitos” representada na Figura 13.3. Note que a restrição *unique* na coluna da direita evita que a tabela associe um livro com mais do que uma editora.

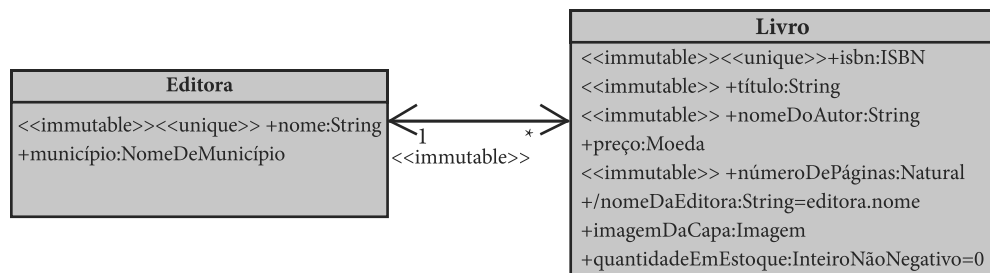


Figura 13.3

Exemplo de associação “um para muitos”.

Tabela 13.4 Tabela associativa que representa uma associação “um para muitos”.	
Tabela: editora_livro	
PK	
Im, NN	Uniq, NN
fkEditora	fkLivro
30002	10001
30001	10002
30002	10003

Conforme visto na Tabela 13.3, a editora 30001 tem um livro (10002) e a editora 30002 tem dois livros (10001 e 10003). Como não se repetem livros nesta tabela, nenhum pode pertencer a mais do que uma editora.

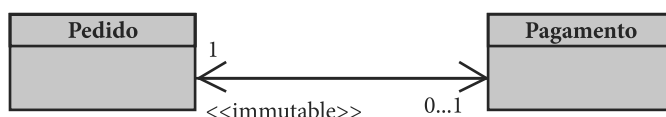
É também possível representar associações de “muitos para um” como *chaves estrangeiras* na tabela que representa a classe do lado *muitos* da associação. Por exemplo, como cada livro pode ter apenas uma editora, então a associação entre livro e editora poderia ser implementada como na Tabela 13.5.

A chave estrangeira *fkEditora* é uma referência direta à chave primária da tabela *Editora*. Se a associação é estritamente para 1, então a coluna da chave estrangeira não pode ser nula. Se a associação fosse 0...1, então a coluna da chave estrangeira não teria essa restrição.

Embora esta abordagem não seja tão homogênea quanto a das tabelas associativas, ela é usualmente preferida pelos designers porque evita a necessidade de implementar uma nova tabela. Uma desvantagem é que, nesse caso, associações “muitos para muitos” seriam implementadas como tabelas associativas e associações “muitos para um” seriam implementadas dentro das tabelas originais. Além disso, se a associação não é obrigatória (se a multiplicidade é 0...1), e relativamente poucos elementos estiverem associados, haverá muitos valores nulos na coluna da chave estrangeira, desperdiçando espaço de armazenamento e degradando a performance. Entretanto, se a associação é estritamente para 1, esta desvantagem não se aplica.

Tabela 13.5 Forma alternativa de implementar uma associação “muitos para um” sem uma tabela associativa.

Tabela: Livro								
PK, Uniq, Im, NN	Uniq, Im, NN	Im, NN	Im, NN	NN	Im, NN		NN	Im, NN
pkLivro	isbn	título	nome Do Autor	preço	número De Páginas	imagem Da Capa	quantidade Em Estoque	fkEditora
10001	0553286587	Rama II	Arthur C. Clarke and Gentry Lee	6,99	466		2	30002
10002	0553293370	Foundation and Empire	Isaac Asimov	5,99	282		3	30001
10003	0671742515	The Long Dark Tea Time of the Soul	Douglas N. Adams	6,99	307		21	30002

**Figura 13.4**

Exemplo de associação “um para um”.

13.2.2.3 Associações “um para um”

Associações “um para um”, obrigatórias ou opcionais, requerem que a tabela associativa tenha uma restrição *unique* em ambas as colunas da chave primária composta, para evitar que qualquer elemento em ambos os lados apareça mais do que uma vez na tabela associativa. A Figura 13.4 mostra um exemplo de associação “um para um” que é obrigatória em um lado e opcional no outro.

A Tabela 13.6 mostra uma tabela relacional que implementa a associação “um para um” da Figura 13.4.

Como o papel é obrigatório para pagamentos, todas as instâncias de *Pagamento* devem aparecer na tabela associativa, mas nem todas as instâncias de *Pedido* devem aparecer, porque o papel não é obrigatório para elas.

Como no caso de associações “muitos para um”, associações “um para um” também podem ser implementadas como chaves estrangeiras em uma das tabelas originais. A coluna da chave estrangeira deve necessariamente ser *unique* nesse caso. Se o papel de associação for também obrigatório, então a coluna da chave estrangeira não pode ser nula.

Tabela 13.6 Tabela associativa que representa uma associação “um para um”.

Tabela: pedido_pagamento	
PK	
Im, Uniq, NN	Uniq, NN
fkPedido	fkPagamento
50001	60001
50003	60002
50005	60003
50011	60004
50016	60005
50021	60006
50030	60007

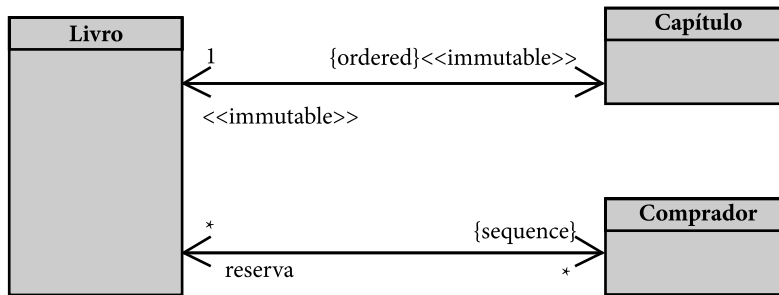


Figura 13.5

Exemplo de papéis ordenados.

13.2.2.4 Associações ordenadas

Uma associação com um papel ordenado (*sequência* ou *conjunto ordenado*) pode ser implementada como uma tabela associativa com uma coluna extra para representar a ordem do elemento na coleção. A Figura 13.5 mostra um exemplo com as duas situações: conjunto ordenado e sequência (uma lista na qual elementos podem se repetir).

A diferença entre a implementação relacional de um conjunto ordenado e uma sequência é que, no caso do conjunto ordenado, a coluna *posição* não deve ser incluída na chave primária composta, como mostrado na Tabela 13.7. Entretanto, no caso de uma sequência (elementos podem se repetir), a coluna *posição* deve ser parte da chave primária composta, a qual, nesse caso, é composta de três colunas (Tabela 13.8).

Tabela 13.7: Tabela relacional que representa um conjunto ordenado.

Tabela: livro_capítulo		
PK		
Im, NN	Im, Uniq, NN	NN
fkLivro	fkCapítulo	posição
10001	130001	1
10001	130002	2
10001	130004	1
10002	130005	2
10003	130006	1
10003	130007	2

Tabela 13.8 Tabela relacional que representa uma sequência.

Tabela: reserva_comprador		
PK		
NN	NN	NN
fkLivro	fkComprador	Posição
10001	20001	1
10001	20003	2
10001	20002	3
10001	20001	4
10002	20003	1
10003	20001	1
10003	20002	2

Na Tabela 13.8, o fato de que a coluna *posição* é incluída na chave primária permite que a mesma pessoa reserve o mesmo livro mais do que uma vez (por exemplo, 20001, Abe, tem duas reservas para o livro 10001, Rama II, na primeira e quarta posições da lista de reservas para este livro). Uma repetição como esta não seria possível na Tabela 13.7, porque a chave primária não inclui a coluna *posição*: o mesmo par *fkLivro/fkCapítulo* não poderia aparecer mais do que uma vez na tabela, não importando sua posição.

Em ambos os casos, para evitar que uma dada posição seja ocupada mais do que uma vez, a restrição *uniq* deveria ser colocada sobre duas colunas: a da classe de origem e a da posição. No caso da Tabela 13.7 e da Tabela 13.8, o par *fkLivro/posição* deveria ser *uniq*.

13.2.2.5 Associações representando bags

No caso de *bags*, nos quais elementos podem se repetir, mas não têm posição, a solução usual é adicionar uma coluna extra à tabela associativa com um contador para o número de vezes que um dado par participa da associação. A Figura 13.6 mostra um exemplo deste tipo de associação.

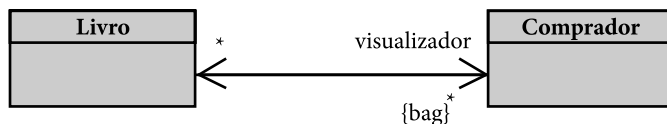


Figura 13.6

Exemplo de um *bag*.

Tabela 13.9 Tabela associativa que representa um *bag*.

Tabela: livro_visualizador		
PK		
NN	NN	NN
fkLivro	fkComprador	Quantidade
10002	20002	1
10001	20002	2
10002	20001	6
10003	20002	1

A Tabela 13.9 mostra a implementação da tabela associativa para o exemplo da Figura 13.6.

A Tabela 13.9 especifica que Abe (20001) visualizou o livro “Foundation and Empire” (10002) seis vezes. Beth (20002) visualizou “Rama II” (10001) duas vezes, “Foundation and Empire” (10002) uma vez e “The Long Dark Tea-Time of the Soul” (10003) uma vez. Não é necessário representar na tabela qualquer par cuja quantidade seja zero; é por isso que Charles (20003) não aparece na tabela: ele nunca visualizou nenhum livro.

13.2.2.6 Associações qualificadas

No caso de uma associação qualificada definida como mapeamento (com multiplicidade 1 ou 0...1) com um qualificador interno (o qualificador é um atributo da classe qualificada), é suficiente implementar a associação como uma simples associação “um para muitos” ou “muitos para muitos” dependendo da multiplicidade no lado do qualificador, como explicado nas seções anteriores.

O único cuidado especial que o designer do banco de dados deve tomar nesse caso é garantir que a coluna do atributo do qualificador seja *unique* e *immutable*. Ela pode também ser indexada para acesso mais rápido aos registros.

Entretanto, quando o qualificador é *externo*, é necessário adicionar uma terceira coluna à tabela associativa para permitir a representação do qualificador. A Figura 13.7 mostra um exemplo desta situação.

A Tabela 13.10 mostra a implementação para o mapeamento definido na Figura 13.7. A tabela associativa tem uma chave primária que é composta apenas da chave estrangeira da classe na origem e do qualificador. A classe de destino é deixada fora da chave primária. Entretanto, ela deve ser marcada como *unique* porque, no exemplo, cada telefone tem um único tipo.

Se o qualificador externo definir uma partição (multiplicidade * no destino), como será mostrado na Figura 13.8, ele é implementado conforme mostrado na Tabela 13.10. Mas, no caso de uma partição com um qualificador externo, a chave primária deve ter três partes, incluindo as chaves estrangeiras de origem e destino, bem como o qualificador. Além disso, como a multiplicidade de origem é 1, a coluna da classe de destino deve ser marcada com *uniq*. No exemplo mostrado na Tabela 13.11, isso significa que um livro não pode ter mais do que um gênero.

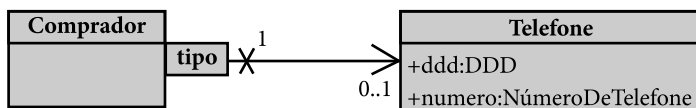


Figura 13.7

Exemplo de um mapeamento com qualificador externo.

Tabela 13.10 Tabela associativa que representa um mapeamento com um qualificador externo.

Tabela: comprador_telefone		
PK		
NN	NN	Uniq, NN
fkComprador	Tipo	fkTelefone
20001	residencial	70001
20001	celular	70002
20002	residencial	70003

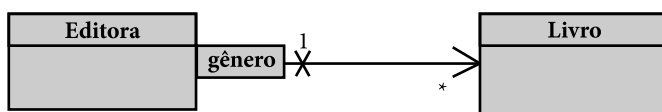


Figura 13.8

Exemplo de uma partição com um qualificador externo.

Tabela 13.11 Tabela associativa representando uma partição com um qualificador externo.

Tabela: editora_livro		
PK		
NN	NN	Uniq, NN
fkEditora	gênero	fkLivro
60001	Ficção	10001
60001	Ficção	10002
60002	Humor	10003

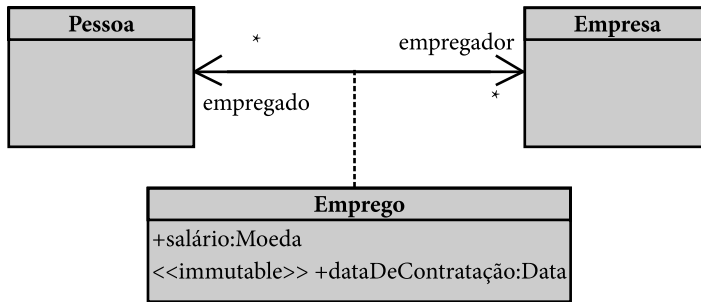


Figura 13.9

Exemplo de uma classe de associação.

Tabela 13.12 Representação relacional de uma associação com classe de associação – Parte 1: A classe de associação.

Tabela: Emprego		
PK, NN, Im, Uniq	NN	NN, Im
pkEmprego	salário	dataDeContratação
80001	1.500,00	15/02/2008
80002	1.200,00	01/03/1999
80003	2.000,00	16/04/2005
80004	900,00	17/01/2001

Tabela 13.13 Representação relacional de uma associação com classe de associação – Parte 2: A associação.

Tabela: empregado_empregador		
PK		
NN	NN	NN, Im, Uniq
fkPessoa	fkEmpresa	fkEmprego
20001	70001	80001
20001	70005	80002
20002	70001	80003
20003	70002	80004

Se a multiplicidade de papel na origem fosse * (definindo uma relação na qual um livro poderia ter mais do que um gênero), a implementação seria basicamente a mesma. A única diferença é que a restrição *unique* na coluna de destino (*fkLivro*) não deveria existir.

13.2.2.7 Classes de associação

Uma associação com classe de associação é representada em duas partes: uma tabela relacional para a classe de associação com seus atributos e uma tabela associativa com uma referência para a tabela da classe de associação.

A Figura 13.9 mostra uma classe de associação e as Tabelas 13.12 e 13.13 implementam seu equivalente relacional.

Assim, uma forma de representar associações com classes de associação é usando uma tabela para representar a classe de associação (Tabela 13.12) e uma tabela associativa com três colunas (Tabela 13.13): as duas primeiras são as chaves estrangeiras para as classes participantes (que formam a chave primária composta da associação), e a terceira é a chave estrangeira para a classe de associação, a qual não é parte da chave primária composta da tabela, mas deve ser imutável e *unique*.

Outra restrição é que todos os valores para *pkEmprego* na tabela *Emprego* devem aparecer na coluna *fkEmprego* da tabela *empregado_empregador*.

13.2.2.8 Associações *n*-árias

No caso de associações entre três ou mais classes (*n*-ária), uma tabela associativa é definida na qual a chave primária é formada pelas chaves primárias de todas as classes participantes. A Figura 13.10 mostra um exemplo de uma associação ternária e a Tabela 13.14 mostra sua representação relacional.

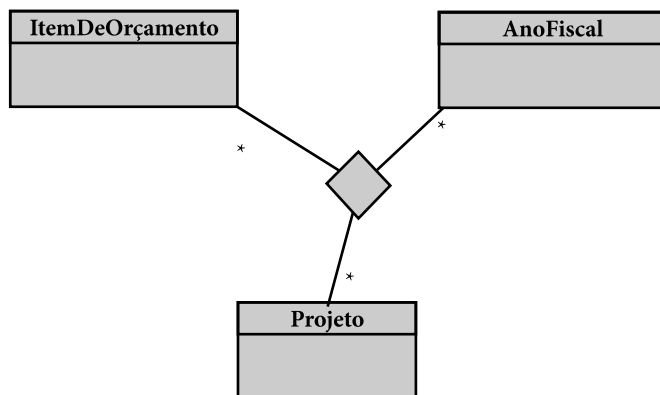


Figura 13.10

Um exemplo de associação ternária.

Tabela 13.14 Equivalente relacional de uma associação ternária.

Tabela: itemDeOrçamento_anoFiscal_projeto		
PK		
NN	NN	NN
fkItemDeOrçamento	fkAnoFiscal	fkProjeto
90001	100001	110001
90001	100002	110002
90002	100001	110003

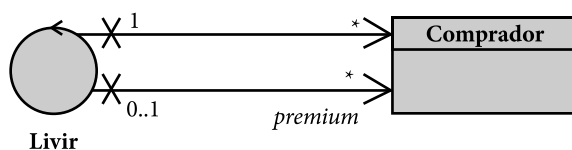


Figura 13.11

Uma associação obrigatória e uma associação opcional para um controlador-fachada.

Na Tabela 13.14, pares como *fkItemDeOrçamento/fkProjeto* podem repetir elementos (90001/100001, por exemplo). Mas a tripla *fkItemDeOrçamento/fkAnoFiscal/fkProjeto* nunca podem se repetir.

13.2.2.9 Associações temporárias e do controlador-fachada

Associações temporárias não são representadas em tabelas relacionais porque, por sua própria definição, elas existem apenas na memória principal e não é necessário nem desejável fazê-las persistir.

Algumas das associações do controlador-fachada também não devem persistir. Associações do controlador para todas as instâncias de uma classe, que sejam obrigatórias para as instâncias, não precisam ser transformadas em tabelas associativas, porque elas sempre repetiriam os mesmos valores para o controlador e teriam todos os elementos da tabela do outro lado. Esta informação já está disponível na coluna da chave primária da tabela que representa a classe conceitual e, portanto, uma eventual tabela associativa seria redundante. Por exemplo, na Figura 13.11, a associação entre o controlador e *Comprador* sem nome de papel explícito é obrigatória para todos os compradores. Assim, uma tabela associativa para representar esta associação conteria apenas as chaves primárias de todos os compradores. Como elas já estão representadas na tabela *Comprador*, repeti-las em uma tabela diferente é desnecessário.

Entretanto, esta observação só é válida se o papel de associação for estritamente para 1 do lado do controlador. Se o lado do controlador tem multiplicidade 0..1, então a associação deveria ser representada separadamente. Na Figura 13.11, a associação com o papel *Premium* deve ser representada porque nem todo comprador pertence a ela.

Nem todo comprador é um comprador *premium*¹⁰, e assim eles devem ser listados em algum lugar. Existem pelo menos duas escolhas para representar compradores *premium*: adicionar um campo booleano à tabela *Comprador* indicando quais compradores são *premium*, ou criar uma tabela de coluna individual que contém as chaves primárias dos compradores *premium*.

A tabela de compradores *premium* não precisa incluir uma coluna para representar a chave primária do controlador porque (1) ele não é uma entidade (e, portanto, não tem chave primária) e, (2) já que o controlador é um singleton e mesmo que uma chave primária fosse atribuída a ele, repetir o mesmo valor em todas as linhas da tabela seria desnecessário.

A Tabela 13.15 mostra uma possível implementação para esta associação opcional.

De acordo com a Tabela 13.15, apenas Abe (20001) e Bea (20002) são compradores *Premium*, e Charles (20003) não é.

13.2.3 Herança

Bancos de dados relacionais não suportam o conceito de herança diretamente. Mapear relações de herança para bancos de dados relacionais é um tópico que requer atenção. Existem várias abordagens diferentes e esta seção discute algumas delas. Todos os exemplos nas subseções seguintes são baseados na Figura 13.12.

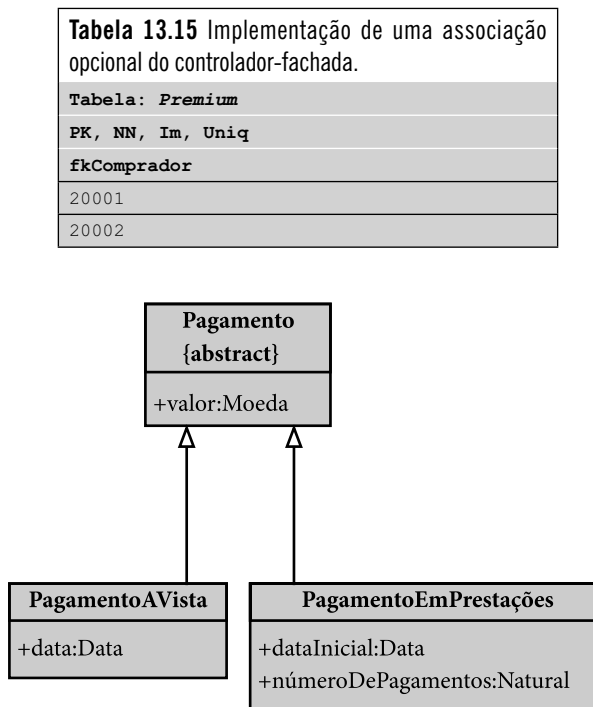


Figura 13.12

Uma situação com herança de atributos.

¹⁰ Note que esta é uma associação normal e não uma associação derivada.

Tabela 13.16 Implementação de herança em uma única tabela com campo *tipo*.

Tabela: Pagamento					
PK, NN, Uniq, Im	NN, Im	NN			
pkPagamento	tipo	valor	data	data Inicial	númeroDe Pagamentos
200001	pagamento AVista	300,00	09/04/2014		
200002	pagamento AVista	251,00	02/07/2014		
200003	pagamento EmPrestações	1.890,00		09/12/2013	12

13.2.3.1 Implementação da hierarquia inteira em uma única tabela

Uma solução para representar herança que parece simples à primeira vista é implementar a hierarquia completa em uma única tabela. Esta tabela contém todos os atributos de todas as classes na hierarquia. Ela também identifica qual classe está sendo representada em cada registro; isso pode ser conseguido pela adição de uma coluna *tipo*, como na Tabela 13.16.

Apenas atributos que pertencem à classe no topo da hierarquia podem ser *não nulo*, porque todos os outros atributos poderiam ser nulos quando o registro pertencer a uma subclasse e o atributo a outra subclasse. Por exemplo, se o tipo de registro é *pagamentoAVista*, então *dataInicial* e *númeroDePrestações* devem necessariamente ser nulos. Se for assumido que um objeto não pode mudar sua classe após a instanciação, então a coluna *tipo* deve ser imutável.

Se herança múltipla for usada (por exemplo, se um pagamento pudesse ser à vista e em prestações ao mesmo tempo)¹¹, então a coluna *tipo* deveria ser trocada por um conjunto de colunas booleanas: *éPagamentoAVista* e *éPagamentoAPrazo*. Nesse caso, um pagamento poderia ser à vista, ou em prestações, ou ainda ambas as coisas ao mesmo tempo (se a classe *Pagamento* não fosse abstrata, ainda haveria a possibilidade de um pagamento não ser nem à vista e nem a prazo).

Esta abordagem tem algumas desvantagens. É difícil gerenciar a consistência entre as subclasses porque todos os dados estão armazenados no mesmo local. Gerenciar isso adequadamente exigiria vários mecanismos de controle complexos. Além disso, vários campos na tabela ficariam nulos o tempo todo e esta grande tabela acabaria sendo bastante esparsa, especialmente se uma hierarquia grande e complexa estiver sendo representada.

13.2.3.2 Cada classe concreta em uma única tabela

Outra abordagem para representar herança é representar cada classe concreta como uma tabela separada. Cada tabela conteria os atributos de uma classe concreta e os atributos de todas as suas superclasses. As Tabelas 13.17 e 13.18 mostram uma possível implementação usando esta abordagem.

¹¹ Embora isso não possa ser verdade na prática; trata-se apenas de uma suposição para ilustrar o exemplo.

Tabela 13.17 Implementação de herança que usa uma tabela para cada classe concreta: *PagamentoAVista*.

Tabela: PagamentoAVista		
PK, NN, Uniq, Im	NN	NN
pkPagamento-AVista	valor	Data
200001	300,00	09/04/2014
200002	251,00	02/07/2014

Tabela 13.18: Implementação de herança que usa uma tabela para cada classe concreta: *PagamentoEmPrestações*.

Tabela: PagamentoEmPrestações			
PK, NN, Uniq, Im	NN	NN	NN
pkPagamento	valor	dataInicial	númeroDePagamentos
200003	1.890,00	09/12/2013	12

Se apenas atributos forem herdados, esta abordagem pode funcionar. Entretanto, se as superclasses definem suas próprias associações, elas precisam ser herdadas, e então gerenciar isso se torna uma dor de cabeça porque ou a tabela associativa terá de obter dados de diferentes tabelas em uma mesma coluna, ou a tabela associativa deverá ser implementada como uma tabela distinta para cada subclasse que herda a associação.

13.2.3.3 Cada classe em uma única tabela

Uma escolha mais adequada para implementar herança, dados os problemas mencionados anteriormente, é definir uma tabela para cada classe da hierarquia, mesmo as abstratas. Dessa forma, herança de atributos e associações podem ser facilmente implementadas. A maior desvantagem é que, ao instanciar um objeto, um número de tabelas igual ao número de superclasses dele mais um deverá ser acessado. As Tabelas 13.19 a 13.21 mostram como implementar esta abordagem. Deve haver referências das tabelas das subclasses para cada tabela de superclasse imediata.

Tabela 13.19 Implementação de herança que usa uma tabela para cada classe: *Pagamento*

Tabela: Pagamento	
PK, NN, Uniq, Im	NN
pkPagamento	Valor
200001	300,00
200002	251,00
200003	1.890,00

Tabela 13.20 Implementação de herança que usa uma tabela para cada classe: *PagamentoAVista*.

Tabela: PagamentoAVista		
PK, NN, Uniq, Im	NN, Uniq, Im	NN
pkPagamentoAVista	fkPagamento	Data
300001	200001	09/04/2014
200002	200002	02/07/2014

Tabela 13.21 Implementação de herança que usa uma tabela para cada classe: *PagamentoEmPrestações*.

Tabela: PagamentoEmPrestações			
PK, NN, Uniq, Im	NN, Uniq, Im	NN	NN
pkPagamentoEmPrestações	fkPagamento	dataInicial	númeroDePrestações
400001	200003	09/12/2013	12

13.3 Salvamento e Carregamento de Objetos

A equivalência entre o design orientado a objetos e o banco de dados relacional é apenas parte da questão de compatibilidade entre estes dois modelos. É necessário também decidir quando os objetos serão carregados e salvos no banco de dados. Alguns designers preferem determinar eles próprios o momento no qual tais operações deveriam ser realizadas. Entretanto, esta abordagem *manual* para salvar e carregar objetos é sujeita a erros de lógica e usualmente ela polui o código da camada de domínio.

Adicionalmente, se o designer é quem vai decidir quando salvar e carregar objetos, algumas vezes essas operações poderão estar sendo executadas desnecessariamente (por exemplo, carregando objetos que já estão em memória e salvando objetos que não foram alterados). Controlar tais aspectos caso a caso, método por método, não é a forma mais produtiva de desenvolver software.

É possível implementar o processo de salvamento e carregamento de objetos com mecanismos automáticos. Nesse caso, o designer deveria apenas decidir quais classes, atributos e associações são persistentes, e um conjunto completo de métodos e estruturas de dados será automaticamente criado para permitir que estes elementos sejam carregados e salvos nos momentos apropriados. Inicialmente, esta seção apresenta a implementação básica ou ingênua para este mecanismo. Mais adiante, as limitações da técnica são explicadas e possíveis soluções propostas.

13.3.1 Proxy virtual

Para implementar um mecanismo automático para salvar e carregar objetos, podemos usar um padrão de design chamado *Proxy virtual* (Gamma; Helm; Johnson; Vlissides, 1995). Um proxy virtual é um objeto muito simples que implementa apenas duas responsabilidades:

- Ele deve conhecer o valor da chave primária do objeto real que ele representa.
- Ele deve redirecionar para o objeto real todas as mensagens que receber em seu nome.

A seguir está o rascunho de como um proxy virtual funciona:

```
CLASSE ProxyVirtual
  VAR pkObjetoReal:ChavePrimária
  PARA QUALQUER mensagem msg recebida FAÇA
    objetoReal:=GerenciadorDeBrokers.get(pkObjetoReal)
    objetoReal.msg()
  FIM PARA
FIM CLASSE
```

Mais adiante, nas seções seguintes, a forma como o *GerenciadorDeBrokers* funciona será gradualmente explicada.

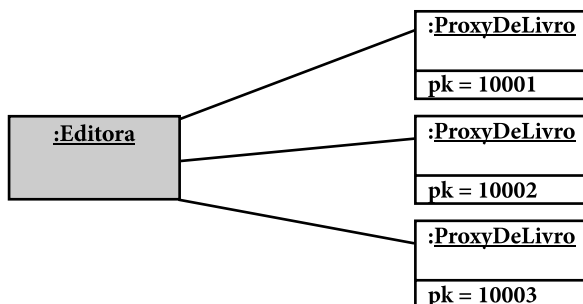
Assim, o design com proxies virtuais requer que, em vez de associar objetos de domínio diretamente a outros objetos de domínio, eles devem ser associados aos seus proxies. Dessa forma, é possível trazer para a memória uma instância de *Editora* sem carregar todas as instâncias de *Livro* que estão ligadas a ela. A instância de *Editora* estará associada com os *proxies de livros*, os quais são objetos muito simples. Os proxies são criados na memória principal e requerem muito menos espaço do qual as instâncias da classe do objeto real, tais como as de *Livro*. Uma instância de *Livro* é carregada apenas se necessário, isto é, apenas se uma mensagem for enviada a ela por meio de seu proxy. Esta forma econômica de usar a memória é chamada de *carregamento preguiçoso*¹², e ela é muito eficiente em termos de uso do tempo e da memória em algumas situações.

Para evitar que o designer tenha de se preocupar sobre quando os objetos devem ser carregados, o mecanismo do proxy virtual deve ser interposto a todas as ligações persistentes na memória principal. Objetos reais enviam mensagens um para o outro como se os proxies não existissem. Mas os proxies interceptam cada mensagem. Os proxies garantem que o objeto real será carregado se ele não estiver na memória.

A Figura 13.13 é um exemplo do mecanismo de carregamento preguiçoso. Inicialmente (Figura 13.13a), apenas uma instância de *Editora* está na memória principal. Ela está associada a três livros. Entretanto, em vez de ter os livros na memória, apenas seus proxies estão lá. Se a instância de *Editora* deve enviar uma mensagem a um dos livros, ela simplesmente envia a mensagem pelo link; a mensagem é interceptada pelo proxy que chama o *GerenciadorDeBrokers*, o qual garante que o livro seja carregado para a memória (Figura 13.13b). Como o livro é associado a alguns capítulos, apenas os proxies dos capítulos são criados em memória, não os objetos reais.

¹² Lazy load.

a)



b)

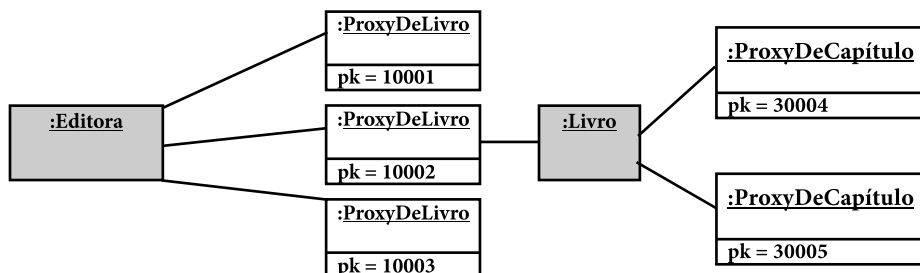


Figura 13.13

(a) Estado inicial no qual apenas uma editora e proxies de livros estão em memória principal; (b) Situação após a editora enviar uma mensagem para um dos livros.

Se um dos capítulos recebe uma mensagem do livro, então apenas o capítulo seria trazido para a memória.

13.3.1.1 Estruturas de dados virtuais

A implementação de proxies virtuais para cada objeto pode ser muito ineficiente quando um objeto tem muitas ligações; por exemplo, uma editora com 50 mil livros registrados poderia requerer a instânciação de 50 mil proxies a serem associados a ela quando ela fosse trazida para a memória. Felizmente, existe uma forma de evitar instanciar grandes quantidades de proxies, a qual consiste na implementação de estruturas de dados virtuais para substituir fisicamente a implementação das associações em memória principal.

Assim, uma editora não teria 50 mil ligações para 50 mil proxies de *Livro*, mas uma única ligação para um *ConjuntoVirtual* com 50 mil PKs. O *ConjuntoVirtual* implementa as operações normais de adição, remoção, atualização e consulta de elementos: as mesmas operações que um conjunto normal deveria implementar. A única diferença é que ele não armazena os objetos reais, mas apenas suas chaves primárias. O conjunto virtual não traz objetos para a memória; ele traz apenas os números de suas chaves primárias. O *ConjuntoVirtual* e suas contrapartes, *SequênciaVirtual*, *ConjuntoOrdenadoVirtual*, *BagVirtual*, *MapeamentoVirtual* etc., usa o *GerenciadorDeBrokers* para carregar objetos reais quando necessário, em vez de manter os objetos reais.

Uma estrutura de dados virtual pode ser implementada a partir dos seguintes princípios:

- Em vez da representação física de uma coleção de objetos, ela é a representação física de uma coleção de valores de chaves primárias para os objetos reais.
- O método que adiciona um objeto na coleção deve adicionar apenas a chave primária do objeto real na representação física.
- O método que remove um objeto da coleção deve apenas remover a chave primária do objeto real da representação física.
- Qualquer método que realize uma consulta sobre a estrutura de dados para retornar um ou mais objetos, recebe o(s) objeto(s) reais(s) que são solicitados ao gerenciador de brokers a partir do valor da chave primária.

Assim, a adição e a remoção de ligações entre objetos podem ser executadas sem que se tenha os objetos reais em memória (pelo menos de um lado da ligação). Um objeto só é trazido para a memória quando a informação sobre ele se torna necessária, ou seja, quando ele recebe uma mensagem.

13.3.1.2 Discussão

O carregamento preguiçoso é útil porque ele só traz para a memória objetos que vão receber uma mensagem. Por exemplo, se uma instância de *Editora* está na memória principal e um de seus livros vai ser atualizado, então não é necessário carregar todas as instâncias de *Livro* que estão ligadas à editora, mas apenas uma.

No entanto, se a instância de *Editora* deve pesquisar seus livros para encontrar o mais caro de todos, então todas as instâncias de *Livro* que estão ligadas a ela deveriam ser carregadas para a memória principal. Esta é a principal falha dessa técnica. Realizar todas as operações sobre objetos na memória principal pode ser uma extraordinária perda de tempo. Imagine carregar 50 mil instâncias de livros para a memória apenas para descobrir qual delas tem o preço mais alto.

Proxies virtuais funcionam bem quando relativamente poucos objetos são trazidos para a memória para cada operação de usuário. Por exemplo, um usuário que pesquisa livros para comprar só vai visualizar uma quantidade relativamente pequena de livros na lista. Este usuário só vai operar sobre um único carrinho de compras e pedido. Este tipo de operação, sobre conjuntos pequenos de objetos, pode ser gerenciado perfeitamente de forma automática pelos proxies virtuais. Entretanto, quando consultas ou comandos envolvem iterar sobre grandes coleções de objetos, como, por exemplo, aumentar o preço de todos os livros da loja, então o uso de proxies virtuais deve ser evitado, a não ser que perda de tempo de processamento não seja um problema; mas este usualmente não é o caso.

Assim, para realizar consultas e comandos sobre grandes coleções de objetos, se a consulta ou comando usa apenas uns poucos atributos dos objetos, seria recomendável considerar substituir o mecanismo de proxy virtual, nesse caso específico, por uma

consulta ou comando diretamente realizado sobre o banco de dados. Portanto, algumas mensagens, quando elas atingem objetos específicos, teriam de ser redirecionadas para uma implementação específica encapsulada, que realiza as ações necessárias no banco de dados e retorna os resultados como desejado, em vez de serem redirecionadas a um proxy.

13.3.2 Brokers e materialização

O processo de carregar um objeto do banco de dados para a memória principal é chamado de *materialização*. Materialização é usualmente solicitada por um proxy para o *gerenciador de brokers*, o qual pode, por sua vez, delegar a materialização para um *broker especializado*. O gerenciador de brokers verifica se o objeto está em memória. Se não estiver, então o gerenciador de brokers ativa o broker especializado para que ele materialize o objeto.

Cada classe pode ter seu próprio broker especializado. Entretanto, um único broker também pode ser implementado para servir a todas as classes. Um broker especializado deve implementar um método chamado *materializar*, que faz o seguinte:

1. Ele cria, na memória principal, uma instância da classe persistente.
2. Ele inicializa os valores dos atributos da nova instância com valores tomados da respectiva linha e coluna do banco de dados.
3. Ele carrega e inicializa as estruturas de dados virtuais que implementam as associações do objeto com as chaves primárias dos respectivos objetos ligados.

Para obter os valores das chaves primárias dos objetos ligados, o broker especializado deve saber quais associações são definidas para o objeto sendo carregado; então, ele pesquisa as ocorrências da chave primária do objeto nas tabelas associativas que implementam aquelas associações. As chaves primárias de outros objetos associados à chave primária do objeto sendo materializado são adicionadas na estrutura de dados virtual que tiver a responsabilidade de manter a respectiva associação.

Por exemplo, *BrokerDeLivro* poderia materializar instâncias e *Livro*, conforme definido na Figura 13.14.

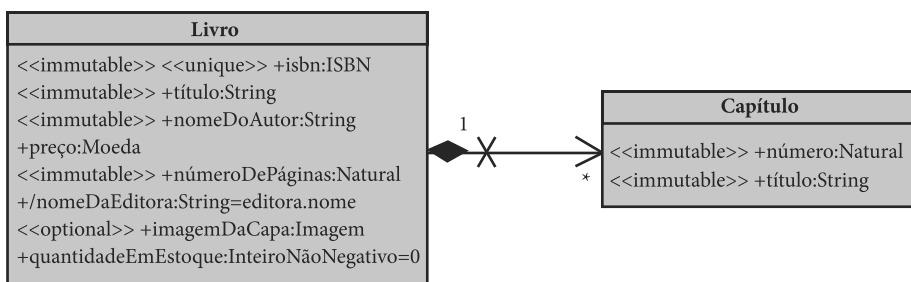


Figura 13.14

Diagrama de classes de referência.

De acordo com este modelo conceitual, *BrokerDeLivro* deve implementar o método *materializar* por meio da realização das seguintes operações:

1. Criar uma instância de *Livro*.
2. Preencher os atributos, *isbn*, *título*, *nomeDoAutor*, *preço*, *númeroDePáginas*, *imagemDaCapa* e *quantidadeEmEstoque* da nova instância com os valores armazenados nas respectivas colunas da tabela *Livro* no banco de dados.
3. Pesquisar a tabela *livro_capítulo* por ocorrências da chave primária do livro na coluna *fkLivro*. Para toda ocorrência, adicionar o valor correspondente encontrado na coluna *fkCapítulo* ao conjunto virtual *capítulo* da nova instância de *Livro*, a qual implementa o papel de associação para o livro.

A materialização realizada pelo broker especializado não deve ser confundida com a criação de uma nova instância, conforme definida pelos contratos de operação de sistema (Capítulo 8). Nesses contratos, a criação de uma instância refere-se à inserção de nova informação no sistema, independentemente de seu armazenamento físico (memória principal ou secundária). A materialização realizada pelo broker apenas se refere à operação de trazer para a memória principal um objeto existente que não está lá ainda. Materialização é, portanto, uma operação que pertence exclusivamente à camada de persistência, não tendo nenhuma relação com a camada de domínio.

13.3.3 Caches

Objetos em memória principal podem ser classificados como segue em relação ao seu estado para com o banco de dados:

- *Limpo* ou *sujo*, dependendo se ele está ou não consistente com a versão armazenada no banco de dados.
- *Velho* ou *novo*, dependendo se ele já existe ou não no banco de dados.
- *Deletado* ou *mantido*, dependendo se ele já foi deletado em memória principal mas ainda não no banco de dados.

Uma *cache* é uma estrutura de dados similar a um *mapeamento* ou *dicionário*, a qual associa valores de chaves primárias com objetos reais representados por elas.

Embora existam oito possíveis combinações das características definidas anteriormente, na prática apenas quatro combinações são suficientes para gerenciar objetos na memória principal:

- *Cache limpa velha*: mantém objetos que estão consistentes com o banco de dados.
- *Cache suja velha*: mantém objetos que existem no banco de dados, mas que foram atualizados em memória principal e estão, portanto, inconsistentes com o banco de dados.
- *Cache nova*: mantém objetos que foram criados em memória principal, mas que ainda não estão no banco de dados.
- *Cache deletada*: mantém objetos que foram deletados em memória, mas que ainda existem no banco de dados.

O gerenciador de brokers verifica se um objeto está em memória principal pela realização de uma consulta nas caches existentes, perguntando pela chave primária do objeto sendo requisitado. Se o gerenciador de brokers encontrar uma referência ao objeto em uma das caches, ele retorna a referência para o proxy que o requisitou. No entanto, se o gerenciador de brokers não encontrar nenhuma referência ao objeto requisitado em nenhuma cache, então ele solicita a um broker especializado (por exemplo, *BrokerDeLivros*) para materializar o objeto. O objeto é então materializado e inserido na *cache limpa velha*.

O mecanismo de persistência deve ter meios de garantir que, toda vez que o estado interno de um objeto na cache limpa velha for alterado, ele seja movido para a cache suja velha. Isso pode ser conseguido, por exemplo, pela adição de um comando especial tal como *GerenciadorDeBrokers.sujar(self)* para qualquer método que atualize o objeto. Se o objeto originalmente estiver na cache limpa velha, este método teria de movê-lo para a cache suja velha.

Objetos que foram criados em memória como resultado de uma pós-condição de contratos são armazenados na *cache nova*. Um objeto que esteja na cache nova não pode ser movido para a cache suja velha, mesmo se seus atributos tiverem sido atualizados. Ele permanece na cache nova até que um *commit* seja realizado. Após o *commit*, ele é movido para uma cache suja velha.

Quando um objeto é deletado na memória principal, o resultado dependerá de qual cache o objeto veio. Se for a cache suja velha ou a cache limpa velha, então ele é movido para a *cache deletada*. Entretanto, se ele estava na *cache nova*, então ele simplesmente é deletado da memória principal.

13.3.3.1 Commit e rollback

As operações de *commit* e *rollback* são usualmente ativadas pela camada de interface para indicar que uma transação teve sucesso e foi confirmada, ou que ela foi cancelada, respectivamente. Essas operações são implementadas pelo gerenciador de brokers. No caso de um *commit*, o gerenciador de brokers deveria fazer o seguinte:

1. Executar uma atualização no banco de dados para todo objeto na *cache suja velha* e mover estes objetos para a *cache limpa velha*.
2. Executar uma inserção no banco de dados para todo objeto na *cache nova* e mover estes objetos para a *cache limpa velha*.
3. Executar uma remoção no banco de dados para todo objeto na *cache deletada*, e deletar esses objetos da memória principal.

No caso do *rollback*, o gerenciador de brokers deve simplesmente remover todos os objetos de todas as caches, exceto aqueles na *cache limpa velha*.

Como a *cache limpa velha* pode crescer indefinidamente, é necessário implementar algum mecanismo para remover os objetos mais antigos toda vez que seu tamanho atingir algum limite previamente estabelecido.

As outras caches podem crescer apenas até o momento em que um *commit* ou *rollback* é realizado. Nesse momento, elas são esvaziadas.

13.3.3.2 Controle de cache em um servidor multiusuário

Se mais do que um usuário se conecta ao sistema, é necessário determinar como compartilhar objetos entre diferentes usuários. Assumindo que se tem uma arquitetura cliente/servidor com camadas de interface, domínio e persistência, pelo menos duas abordagens são possíveis:

- *As três camadas são executadas no cliente.* Não existe memória principal compartilhada no servidor, apenas o banco de dados. O cliente é pesado e o servidor é usado apenas para acesso e armazenamento de dados quando um *commit* for realizado. Nesse caso, o que trafega pela rede são os dados na forma relacional e instruções SQL¹³. A informação viaja apenas quando um objeto deve ser materializado ou comitado. A desvantagem deste design é que o nodo cliente é sobrecarregado. Entretanto, aplicações cliente têm recentemente se tornado mais ricas e mais complexas: não depender de um servidor para processar a lógica de aplicação pode ser um requisito crucial.
- *As camadas de domínio e persistência são implementadas no servidor, e a camada de interface é implementada no cliente.* Nesse caso, os objetos existem na memória principal apenas no servidor, e o que trafega na rede são os parâmetros das operações de sistema e os retornos das consultas. A vantagem é que clientes são mais leves e usualmente é mais barato atualizar um servidor do que atualizar milhares de clientes. Entretanto, para algumas aplicações que requerem rápido processamento de dados, esperar por um servidor pode não ser viável.

Se os objetos estão fisicamente no servidor apenas, existem ainda mais possibilidades. Em um caso, todos os usuários compartilham as quatro caches, com a desvantagem de que um usuário poderia ter acesso a objetos que estão sendo modificados, mas que ainda não foram comitados por outro usuário. Esta opção não parece ser recomendável para a maioria das aplicações.

A outra opção é compartilhar apenas os objetos na *cache limpa velha* entre os usuários. Deveria haver múltiplas instâncias das outras caches que fossem privativas de cada usuário. Se um objeto está em uma cache privativa de um usuário, então os outros usuários não podem acessá-lo. Eles devem esperar por um *commit* ou *rollback* do primeiro usuário. Assim, o mecanismo de persistência para acesso multiusuário poderia ser implementado da seguinte forma:

- Uma cache limpa velha é compartilhada por todos os usuários.
- Cada usuário tem uma cache suja velha, uma cache nova e uma cache deletada.

¹³ *Structured Query Language*, a linguagem dominante para definição, acesso e atualização de bancos de dados relacionais (Date, 1982).

Fazendo assim, é possível garantir que nenhum usuário tenha acesso a objetos que estejam sendo modificados por outros usuários. Portanto, é possível usar as caches para implementar um mecanismo de *tranca*¹⁴, ou seja, quando um usuário estiver atualizando um objeto, os outros usuários não podem acessá-lo. Apenas quando o usuário que possui o objeto realizar um *commit* ou *rollback*, e o objeto for movido para a cache limpa velha ou removido da memória, os outros usuários poderão ganhar acesso a ele novamente.

Uma vantagem desse método é o uso otimizado de memória principal no servidor. Todos os usuários compartilham os objetos na cache limpa velha, a qual é a única que pode crescer indefinidamente. As outras quatro caches, que são específicas de cada usuário, apenas crescem durante a transação; quando um *commit* ou *rollback* acontece, todas elas são esvaziadas.

Entretanto, isso poderia afetar negativamente a escalabilidade, porque alguns usuários poderiam ficar trancados enquanto outros estivessem lidando com certos objetos. Seria possível implementar um mecanismo de controle mais sofisticado baseado em *junções otimistas*¹⁵: dois ou mais usuários podem editar o mesmo objeto, desde que eles não atualizem o mesmo atributo ou associação.

Isso deve ser feito com cuidado, entretanto. Questões de concorrência são frequentemente preocupações complexas, e muitas decisões podem depender de regras de negócio. Por exemplo, o preço de um livro pode ser atualizado enquanto um usuário estiver finalizando seu pedido?

Algumas aplicações poderiam mesmo requerer uma estratégia de tranca mais pessimista: se um usuário estiver visualizando um objeto, nenhum outro usuário pode sequer ter acesso para visualizá-lo. Este é o caso em aplicações bancárias, por exemplo, as quais requerem o máximo de segurança de dados.

13.4 O processo inteiro

Concepção	Elaboração
Modelagem de negócio	Construir uma visão geral do sistema, incluindo na medida do necessário: • um diagrama de casos de uso de negócio e determinar o escopo de automação para o projeto. • diagramas de atividades de negócio preliminar para os casos de uso de negócio. • diagramas de máquina de estados para objetos-chave de negócio.

¹⁴ Lock.

¹⁵ Optimistic merges.

continuação		
	Concepção	Elaboração
Requisitos	Preparar o diagrama de casos de uso de sistema (requisitos funcionais): • Identificar os atores de sistema a partir do modelo de casos de uso de negócio. • Identificar os casos de uso de sistema a partir do modelo de casos de uso de negócio e dos diagramas de atividades e de máquina de estado do modelo de negócio. Identificar os requisitos não funcionais como anotações aos casos de uso: • Identificar as principais regras de negócio associadas aos casos de uso. • Identificar os principais aspectos de qualidade relacionados aos casos de uso. • Identificar requisitos suplementares.	Detalhar os requisitos, expandindo os casos de uso: • Identificar o fluxo principal. • Identificar os fluxos alternativos: variantes e tratadores de exceção
Análise e design	Preparar o modelo conceitual preliminar a partir da observação dos casos de uso de sistema e dos conceitos que eles necessitam.	Elaborar o diagrama de sequência de sistema: • Representar o fluxo principal de um caso de uso como um diagrama de sequência de sistema. • Representar os comandos e as consultas de sistema, usando a estratégia <i>stateful</i> ou <i>stateless</i>. • Completar o diagrama de sequência de sistema com fluxos alternativos. Refinar o modelo conceitual: • Identificar conceitos, atributos e associações no texto dos casos de uso expandidos. • Detalhar atributos e associações com estereótipos, multiplicidade e restrições na medida do necessário. • Organizar o modelo com o uso de herança, classes de associação e especificações temporais. • Adicionar invariantes na medida do necessário. • Melhorar o modelo conceitual com a aplicação de padrões de análise. Escrever contratos de operação de sistema para comandos e consultas existentes nos diagramas de sequência de sistema: • Identificar precondições e exceções baseando-se nos parâmetros inválidos e restrições complementares. • Identificar pós-condições para comandos e retornos para consultas.

continuação	
Concepção	Elaboração
	Fazer o design da camada de domínio: • Criar diagramas de sequência ou comunicação para cada contrato de comando de sistema. • Usar este diagrama para decidir quais métodos implementar em cada classe. • Inspeccionar os diagramas para descobrir quais associações podem ser unidirecionais e defini-las como tal. • Observar os contratos de consulta de sistema e decidir quais consultas delegadas deveriam ser implementadas; algumas delas poderiam ser atributos ou associações derivados. • Produzir ou refinar o diagrama de classes de design. Fazer o design da camada de interface: • Fazer o design da organização da interface em nível mais alto baseado em áreas e páginas. • Refinar páginas, incluindo novos componentes para parâmetros ou operações de sistema e resultados para consultas de sistema. • Associar a ativação de operações com eventos de interface, como o acionamento de botões ou menus. Fazer o design do mecanismo de persistência: • Reusar ou construir um mecanismo de persistência baseado em ORM, proxies virtuais, caches e brokers. • Decidir onde aplicar o mecanismo de persistência <i>default</i> e onde implementar o acesso direto ao banco de dados para otimização de performance.
Implementação	Gerar código: • Para classes. • Para atributos. • Para associações. • Necessário para métodos básicos como <i>get</i>, <i>set</i>, <i>add</i>, <i>remove</i>, <i>Create</i>, <i>destroy</i> e outros. • Para operações de sistema e métodos delegados usando os modelos dinâmicos como referência.
Teste	Planejar e executar o teste: • Realizar testes de unidade para métodos e classe que não foram gerados automaticamente. • Realizar testes de operação de sistema baseados em contratos de operação de sistema. • Realizar testes de caso de uso baseados em cenários de caso de uso.

Posfácio

A esta altura o leitor deve estar se perguntando por que a imagem de um tear aparece na capa deste livro. Mesmo que você não esteja se perguntando, eu vou contar. A razão tem duas partes: uma simbólica e uma histórica.

A razão histórica é que existem similaridades entre o processo de produção de software e o de produção de tecidos em um tear. Primeiramente, você planeja e faz um rascunho do que tenciona produzir; e então executa seu trabalho e verifica se ele está indo bem. As várias cores dos fios usados para produzir um tecido podem ser comparadas aos vários aspectos da informação e tecnologia que precisam ser tecidos para produzir os *padrões* que são desejados no produto. O trabalho feito em um tear pode ser desfeito, assim como o trabalho feito com software. Após isso, ele pode ser refatorado.

Produzir belos padrões em tecidos pode parecer complexo e confuso para pessoas leigas; o mesmo é verdade para a produção de software. Mas se as melhores práticas, técnicas e ferramentas forem usadas, o processo é facilitado.

Uma diferença entre estes processos é que trabalhar com teares foi automatizado desde o século XVIII, enquanto a indústria de software está dando os primeiros passos na direção da automatização recentemente. Além disso, o desenvolvimento de software é tão complexo que muitas pessoas duvidam que ele possa ser automatizado com sucesso algum dia. Entretanto, embora a complexidade do software cresça a cada ano, novas técnicas para melhorar nossa capacidade de lidar com esta complexidade estão sendo aprendidas e usadas.

A razão histórica que liga teares a este livro baseia-se nos seis graus de separação:

1. Uma das primeiras máquinas automáticas complexas foram os *teares* controlados por *cartões perfurados* que foram inventados na França por volta de 1725 e bastante melhorados por Joseph Marie Jacquard em 1801. Eles se tornaram um grande sucesso comercial e quase custaram a vida de Jacquard por conta da furiosa oposição dos tecelões de seda que temiam perder seus empregos.
2. *Cartões perfurados* foram usados por *Herman Hollerith* como mecanismo de entrada de dados para a sua Máquina de Tabulação Elétrica, a qual foi usada no censo americano de 1880, reduzindo drasticamente o tempo para obter os resultados.
3. A companhia de *Hollerith*, a Tabulating Machine Company fundiu-se com outras empresas em 1911 para formar a Computing Tabulating Recording Company (CTR), a qual em 1924 trocou seu nome para *International Business Machines Corporation (IBM)*.

4. Em 2003, a IBM adquiriu por 2,3 bilhões de dólares a empresa Rational, a qual foi responsável, entre outros produtos, pelo *Processo Unificado* e pela *Linguagem de Modelagem Unificada (UML)*.
5. Este *livro* trata de vários aspectos do *Processo Unificado* e usa *UML* e seus subprodutos como OCL e IFML para modelagem.

Até onde posso ver, cinco graus de separação foram suficientes.

Raul Sidnei Wazlawick, 12 de setembro de 2013 (traduzido pelo autor em 30 de maio de 2014).

Referências

- Adams, D. N. (1979). *The Hitchhiker's Guide to the Galaxy*. Completely Unexpected Productions Ltd.
- Albrecht, A. J. (1979). Measuring application development productivity. *Proceedings of the joint SHARE/GUIDE and IBM Application Development Symposium*. (p. 83-92). Chicago.
- Albrecht, A. J., & Gaffney Jr., J. E. (1983, November). Software Function, Source Lines of Code, and Development Effort Prediction: A software science validation. *IEEE Transactions on Software Engineering*, 9(6), p. 639-648.
- Alford, M. (1991). *Requirements-Driven Software Design*. McGraw-Hill.
- Ambler, S. W. (2000). *Web Services Programming Tips and Tricks: Modeling essential use cases*. Disponível em: <<http://www.ibm.com/developerworks/library/ws-tip-essentialuse/index.html>>. Acesso em: 31.ago.2013.
- Ambler, S. W. (2004). *The Object Primer: Agile model-driven development with UML 2.0*. 3rd edition. Cambridge University Press.
- Ambler, S. W., & Jeffries, R. (2002). *Agile Modeling: Effective practices for eXtreme Programming and the Unified Process*. John Wiley and Sons.
- Anda, B. (2001). Estimating Software Development Effort Based on Use Cases: Experiences from industry. *4th International Conference on the Unified Modelling Language* (p. 487-502). Toronto: Springer-Verlag.
- Beck, K. (1989). *Simple Smalltalk Testing with Patterns*. Disponível em: <<http://www.xprogramming.com/testfram.htm>>. Acesso em: 31.ago.2013.
- Beck, K. (2003). *Test-Driven Development by Example*. Addison-Wesley.
- Beck, K., & Andres, C. (2004). *Extreme Programming Explained: Embrace change* (2. ed.). Addison-Wesley.
- Beizer, B. (1990). *Software Testing Techniques* (2. ed.). Van Nostrand Reinhold.
- Bloch, A. (1980). *Murphy's Law, and Other Reasons why Things go Wrong*. Los Angeles: Price/Stern/Sloan Publishers, Inc.
- Boehm, B. W. (2000). *Software Cost Estimation with COCOMO II*. Prentice-Hall.
- Booch, G., Maksimchuk, R. A., Engle, M. W., Young, B. J., Conallen, J., Houston, K. A. (2007). *Object-Oriented Analysis and Design with Applications* (3. ed.). Addison-Wesley Professional.
- Bourret, R. (2010). *XML Database Products*. Disponível em: <<http://www.rpbourret.com/xml/XMLDatabaseProds.htm>>. Acesso em: 11.set. 2013.
- Braz, M., & Vergilio, S. (2006). Software Effort Estimation Based on Use Cases. *Proceedings of the 30th Annual International Computer Software and Applications Conference*, (p. 221-228).
- Burnstein, I. (2003) *Practical Software Testing*. Springer-Verlag.

- Cabot, J. (2007). From Declarative to Imperative UML/OCL Operation Specifications. *Lecture Notes in Computer Science*, p. 198-213.
- Cardoso, I. S. (2011). *Inserindo Suporte a Declaração de Associações da UML 2 em uma Linguagem de Programação Orientada a Objetos*. Masters Dissertation. UFSC – Federal University of Santa Catarina, PPGCC – Computer Science. Florianópolis, Brazil.
- Carr, M. J., Konda, S. L., Monarch, I., Ulrich, F. C., & Walker, C. F. (1993). *Taxonomy-Based Risk Identification*. SEI. Pittsburgh: Carnegie Mellon University.
- Center for Software Engineering USC (2000). *COCOMO II - Model definition manual - version 2.1*. Disponível em: <<http://csse.usc.edu/csse/TECHRPTS/2000/usccse2000-500/usccse2000-500.pdf>>. Acesso em: 31.ago.2013.
- Ceri, S., Fraternali, P., Bongio, A., Brambilla, M., Comai, S., & Matera, M. (2003). *Designing Data-Intensive Web Applications*. Morgan Kaufmann.
- Choenni, S., Blanken, H., Chang, T. (1993). Index Selection in Relational Databases. *Proc. International Conference on Computing and Information*. 491-496.
- Cockburn, A. (2001). *Writing Effective Use Cases*. Addison-Wesley.
- Crain, A. (2004). *RUP Iteration Planning*. IBM. Disponível em: <<http://www.ibm.com/developerworks/rational/library/5335.html>>. Acesso em: 15.set. 2013.
- Date, C. J. (1982). *An Introduction to Database Systems*. Addison-Wesley.
- Dustin, E. (2002). Automate Regression Tests when Feasible. In E. Dustin, *Effective Software Testing: 50 specific ways to improve your testing*. Addison-Wesley Professional.
- Emam, K. E., Melo, W., & Drouin, J.-N. (1997). *Spice: The Theory and Practice of Software Process Improvement and Capability Determination*. Los Alamitos, CA, USA: IEEE Computer Society Press.
- English, A. V. (2007). *Business Modeling with UML: Understanding the similarities and differences between business use cases and system use cases*. Acesso em: <<http://www.ibm.com/developerworks/rational/library/apr07/english/>>. Acesso em: 14.nov.2012.
- Fowler, M. (2003). *Patterns of Enterprise Application Architecture*. Addison-Wesley.
- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of reusable object-oriented software*. Addison-Wesley.
- Gause, D. C., Weinberg, G. M. *Exploring Requirements: Quality before design*. Dorset House Pub.
- Goldberg, A., & Robson, D. (1989). *Smalltalk 80: The language*. Addison-Wesley.
- Grady, R. (1992). *Practical Software Metrics for Project Management and Process Improvement*. Prentice-Hall.
- Ibarra, G., & Vilain, P. (2010). Software Estimation Based on Use Case Size. *Brazilian Symposium on Software Engineering*, (p. 178-187). Florianópolis.
- Ireland, C., Keynes, M. Bowers, D., Newton, M., Waugh, K. (2009). A Classification of Object-Relational Impedance Mismatch. *Advances in Databases, Knowledge, and Data Applications, DBKDA '09*. p. 36-43. Gosier.
- Jacobson, I. (1994). *The Object Advantage: Business process reengineering with object technology*. Addison-Wesley.

- Jacobson, I., Booch, G., & Rumbaugh, J. (1999). *The Unified Software Development Process*. Addison-Wesley.
- Jacobson, I., Christenson, M., Jonsson, P., & Övergaard, G. (1992). *Object-Oriented Software Engineering: A use case driven approach*. Addison-Wesley.
- Jacobson, I., Spence, I., & Bittner, K. (2011). *Use Case 2.0: The guide to succeeding with use cases*. Ivar Jacobson International.
- Jones, C. (2007). *Software Estimating Rules of Thumb - version 3*. Disponível em: <<http://www.compaid.com/caiinternet/ezine/capers-rules.pdf>>. Acesso em: 31.ago.2013.
- Kamal, M. W., & Ahmed, M. A. (2011). A Proposed Framework for Use Case Based Effort Estimation using Fuzzy Logic: Building upon the outcomes of a systematic literature review. *International Journal on New Computer Architectures and their Applications*, 1(4), p. 976-999.
- Karner, G. (1993). *Use Case Points: Resource estimation for objectory projects*. Objective Systems.
- Krasner, G. E., Pope, S.T. (1988). A Cookbook for Using the Model-View Controller User Interface Paradigm in Smalltalk-80. *The Journal of Object Technology – JOT*. SIGS Publications. August-September.
- Kroll, P., & Kruchten, P. (2003). *The Rational Unified Process Made Easy: A practitioner's guide to RUP*. Addison Wesley.
- Kroll, P., & MacIsaac, B. (2006). *Agility and Discipline Made Easy: Practices from OpenUp and RUP*. Addison-Wesley Professional.
- Kroll, P. (2004). *Planning and Estimating a RUP Project using IBM Rational SUMMIT Ascendant*. Disponível em: <<http://www.ibm.com/developerworks/rational/library/4772.html>>. Acesso em: 31.ago.2013.
- Kruchten, P. (2003). *The Rational Unified Process: An introduction*. 3. ed. Addison-Wesley.
- Larman, C. (2004). *Applying UML and Patterns: An introduction to object-oriented analysis and design and the unified process* (3. ed.). Prentice-Hall.
- Lieberherr, K., & Holland, I. (1989, September). Assuring Good Style for Object-Oriented Programs. *IEEE Software*, p. 38-38.
- Liskov, B. (1974). Programming with Abstract Data Types, in *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*, p. 50-59, Santa Monica, California.
- McConnell, S. (1996). *Rapid Development*. Microsoft Press.
- Meszaros, G. (2007). *xUnit Test Patterns: Refactoring test code*. Addison-Wesley.
- Meyer, B. (1988). *Object-Oriented Software Construction*. Prentice Hall.
- Miles, R., & Hamilton, K. (2006). *Learning UML 2.0*. O'Reilly.
- Mohagheghi, P., Anda, B., & Conradi, R. (2005). Effort Estimation of Use Cases for Incremental Large Scale Software Development. *International Conference on Software Engineering*.
- Molyneaux, I. (2009). *The Art of Application Performance Testing: Help for programmers and quality assurance*. O'Reilly Media.
- Myers, G. J., Sandler, C., Badgett, T., & Thomas, T. M. (2004). *The Art of Software Testing* (2nd ed.). New Jersey: John Wiley & Sons.
- Nelson, W. B. (2004). *Accelerated Testing: Statistical models, test plans, and data analysis*. John Wiley & Sons.

- Nielsen, J. (1994). *Usability Engineering*. Academic Press.
- Object Management Group. (2010). *OCL 2.3.1 Specification*. OMG.
- Object Management Group. (2011). *OMG Unified Modeling Language™ (OMG UML), Superstructure, Version 2.4.1*. OMG.
- Ockhan, W. (1495). *Quaestiones et decisiones in quattuor libros Sententiarum Petri Lombardi*. Johannes Trechsel.
- Potel, M. (1996). *MVP: Model-View-Presenter: The Taligent Programming Model for C++ and Java*. Disponível em: <<http://www.wildcrest.com/Potel/Portfolio/mvp.pdf>>. Acesso em: 01.set.2013.
- Pressman, R. S. (2008). *Web Engineering: A practitioner's approach*. McGraw Hill.
- Pressman, R. S. (2010). *Software Engineering: A practitioner's approach*. McGraw Hill.
- Probasco, L. (2001). *What makes a good Use Case Name?* The Rational Edge. March.
- Ribu, K. (2001). *Estimating Object-Oriented Software Projects with Use Cases*. Master of Science Thesis, University of Oslo, Department of Informatics.
- Robiolo, G., & Orosco, R. (2008). Employing Use Cases to Early Estimate Effort with Simpler Metrics. *Innovations in Systems and Software Engineering*, 4 (1), p. 31-43.
- Royce, W. W. (1970). Managing the Development of Large Software Systems. *Proceedings of the IEEE WESCON* (p. 1-9). Los Angeles: IEEE.
- Santos, C. C. (2007). *Geração Automática de Diagramas de Comunicação a partir de Contratos OCL*. Masters Dissertation, UFSC, Florianópolis.
- Satzinger, J. W., Jackson, R. B., & Burd, D. S. (2011). *Systems Analysis and Design in a Changing World* (6th ed.). Course Technology.
- Schneider, G., & Winters, J. P. (1998). *Applying Use Cases: A practical guide*. Addison Wesley.
- Scott, K. (2001). *The Unified Process Explained*. Addison-Wesley.
- SEI-CMU. (2010). *CMMI for Development version 1.3*. Carnegie Mellon University.
- Sommerville, I. (2006). *Software Engineering* (7. ed.). Addison-Wesley.
- Symons, C. R. (1988). Function Points Analysis: Difficulties and improvements. *IEEE Transactions on Software Engineering*, 14 (1), p. 2-11.
- Tidwell, J. (2005). *Designing Interfaces: Patterns for effective interaction design*. O'Reilly Media.
- UKSMA Metrics Practices Committee (1998). *Mk II Function Point Analysis Counting Practice Manual – Version 1.3.1*. Disponível em: <<http://uksma.co.uk/products.asp> upon registration>. Acesso em: 31.ago.2013.
- Warmer, J., & Keppe, A. (1998). *The Object Constraint Language: Precise modeling with UML*. Addison-Wesley.
- Wazlawick, R.S. (2013). *Engenharia de Software: Conceitos e práticas*. Elsevier.
- West, D. (2002). *Planning a Project with the Rational Unified Process*. Rational White Paper. Disponível em: <<http://www.nyu.edu/classes/jcf/CSCI-GA.2440-001/handouts/Planning-ProjWithRUP.pdf>>. Acesso em: 15.set.2013.
- Won, K. (1990). *Introduction to Object-Oriented Databases*. The MIT Press.
- Woolf, B. (1998). Null Object. In: Martin, Robert; Riehle, Dirk; Buschmann, Frank. *Pattern Languages of Program Design 3*. Addison-Wesley.